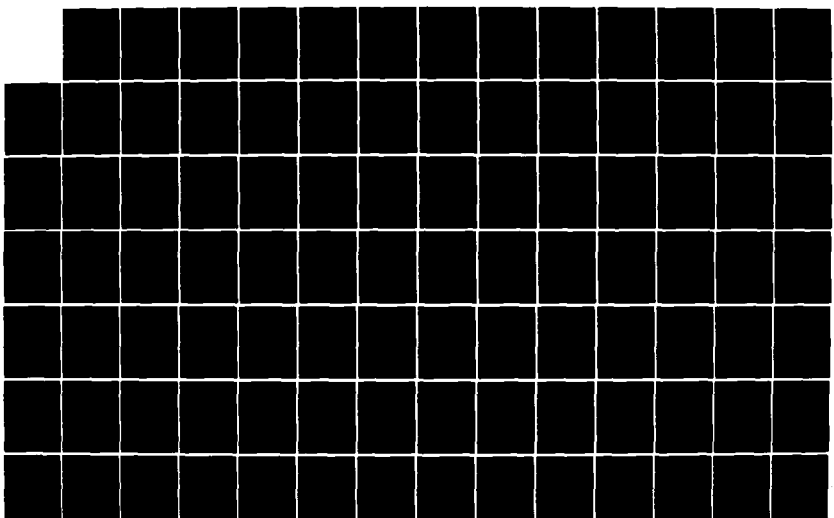
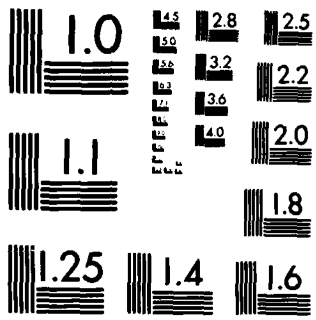
AD-A151 501 DESIGN AND ANALYSIS OF A COMPLETE RELATIONAL INTERFACE 1/2
FOR A MULTI-BACKEND DATABASE SYSTEM(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA R E ROLLINS JUN 84
UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

DESIGN AND ANALYSIS OF A COMPLETE RELATIONAL INTERFACE FOR A MULTI-BACKEND DATABASE SYSTEM

by

Richard Edward Rollins

June 1984

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A151 501 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Edward Rollins	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	12. REPORT DATE June, 1984
	13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

I. Info.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database management systems, multi-backend data system, attribute-based data language, relational data language, relational interface, database kernel Comput. Program.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Organizations of all types are becoming increasingly dependent on the operation of database management systems based on one of the three generally known data models (i.e., network, hierarchical or relational) for the centralized control of operational data. As an alternative to the development of separate, stand-alone systems for specific models, recent research has proposed a system designed to support multiple data models and model-based languages as if the system is a heterogeneous collection of→(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

database systems. This proposal is based on the existence of a simple and powerful data model to which the three well-known models can be mapped. This model, the attribute-based data model, is the data model upon which the Multi-Backend Database System (MDBS), a software database machine, is based. This thesis concentrates on the language interface aspects of implementing MDBS as a kernel for the support of relational databases. In particular, this thesis provides the design and analysis of an interface between the relational query language (SQL) and the attribute-based data language (ABDL). Originator-supplied keywords included:

to front p.

Approved for Public Release; Distribution Unlimited

Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System

by

Richard Edward Rollins
Commander, United States Navy
B.S., United States Naval Academy, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POST GRADUATE SCHOOL
June, 1984

Author: _____

Approved by: _____
Thesis Advisor

_____
Co-Advisor

_____
Chairman, Department of Computer Science

_____
Dean of Information and Policy Sciences

ABSTRACT

Organizations of all types are becoming increasingly dependent on the operation of database management systems based on one of the three generally known data models (i.e., network, hierarchical, or relational) for the centralized control of operational data. As an alternative to the development of separate, stand-alone systems for specific models, recent research has proposed a system designed to support multiple data models and model-based languages as if the system is a heterogeneous collection of database systems. This proposal is based on the existence of a simple and powerful data model to which the three well-known models can be mapped. This model, the attribute-based data model, is the data model upon which the Multi-Backend Database System (MDBS), a software database machine, is based. This thesis concentrates on the language interface aspects of implementing MDBS as a kernel for the support of relational databases. In particular, this thesis provides the design and analysis of an interface between the relational query language (SQL) and the attribute-based data language (ABDL).

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The work reported in this thesis is part of ongoing research efforts conducted by the Laboratory for Database Systems Research, Department of Computer Science, Naval Postgraduate School, Monterey, California, 93943. The laboratory is under the direction of Dr. D. K. Hsiao. This work is supported by Contract N00014-84-WR-24058 from the Office of Naval Research and by an equipment grant from the External Research Program of the Digital Equipment Corporation.

I would like to extend my gratitude to the following people:

Dr. David K. Hsiao, for the opportunity, guidance, and most importantly, for the motivation.

Dr. Paula R. Strawser, for her professionalism, detailed guidance, and "gentle prodding".

My wife, Shirl, and my children, Jenni and Jamie, for their support and understanding.

I. INTRODUCTION

Database technology is rapidly becoming an extremely important aspect of data processing. Commercial database management systems have only been available since the 1960's. Today, many thousands of organizations (e.g., corporations, universities, governments) are critically dependent on the efficient and reliable operation of these systems. Each of these organizations has invested large amounts of time, energy, and money to ensure that the various end users are provided the data they need for doing their jobs as effectively and efficiently as possible. Any of the three generally known approaches to the design of database systems (i.e., network, hierarchical, and relational) provides for the centralized control of an organization's operational data. However, questions concerning the ease of understanding, use, and implementation have stimulated research to determine the "best" approach. The earliest database systems were based on the network or the hierarchical model. These models lend themselves well to the efficient implementation necessary for the maintenance of large databases. Today, with the increased emphasis on the ease of use and understanding, many of the newer commercialized systems are based on the relational model. Examples of commercially available

systems based on these models include: IMS (hierarchical), SQL/DS (relational), and IDMS (network). Each of these systems utilizes a model-based data language which allows the user to specify the operations to be performed on the data.

Once a commitment is made to manage a large database containing an organization's operational data through the implementation of one of these systems, it is financially prohibitive to change to another approach. In addition to the obvious re-programming requirement, user personnel (including high-level executive users) must be re-trained in the syntax and semantics of a different data language. Demurjian, et. al., have proposed an attractive alternative to the development of separate, stand-alone systems for specific models. Their research, reported in [Ref. 1], proposes that a system can be designed "...to support multiple data models and model-based languages as if the system is a heterogeneous collection of database systems."

The above proposal is based on the existence of a simple and powerful data model to which the network, hierarchical, and relational models can be mapped. This is the attribute-based data model as originally described by Hsiao [Ref. 2] and extended by Wong [Ref. 3]. This is the data model of the Multi-backend Database System (MDBS), a software database system designed by Menon and Hsiao [Ref. 4]. The proposal of [Ref. 1] is that the attribute-based system

(MDBS), with the attribute-based data model and the attribute-based data language (ABDL), can serve as a kernel for the support of several data models and the data languages based on those models.

The attribute-based system is ideally suited to its proposed role as a kernel of database systems. As demonstrated by Banerjee [Refs. 5, 6, and 7], a relational, hierarchical, or network database can be converted into an attribute-based database. The primary database and aggregate operations, RETRIEVE, INSERT, DELETE, UPDATE, MIN, MAX, SUM, COUNT, and AVG are supported by the system's high-level data language, ABDL. Finally, language interfaces can be developed to translate relational, hierarchical, or network data language constructs into ABDL constructs. In this thesis, we are concerned with the language interface aspects of this research.

In particular, this thesis provides the design and analysis of a relational interface to the attribute-based system (MDBS). We extend the work of Macy [Ref. 8], who has shown that a subset of the relational model-based data language, SQL (Structured Query Language) can be directly supported by MDBS and ABDL. Macy has provided mappings from the SQL SELECT, INSERT, DELETE, and UPDATE constructs to the corresponding ABDL constructs: RETRIEVE, INSERT, DELETE, and UPDATE. The translations are limited to queries involving simple, single-relation operations. Using these

basic mappings as a foundation, we show that SQL queries involving set membership operations can also be mapped directly to ABDL constructs. We also demonstrate that other SQL constructs (of particular importance, the nested SQL SELECT) can be mapped to a series of ABDL operations. Finally, we propose a software structure to facilitate the implementation of a complete relational interface for the attribute-based kernel (i.e., MDBS). In the following two sections, we discuss our design goals and our unconventional approach to the design of the SQL interface. In the last section of this chapter, the organization of the thesis is presented.

A. DESIGN GOALS

We are motivated to design a SQL interface to MDBS in order to demonstrate the feasibility of utilizing the attribute-based system as the kernel of database systems in general. However, our intention is not to propose changes to MDBS itself. Instead, we propose that the SQL interface be implemented on the host computer. All translations are accomplished in the SQL interface. MDBS continues to receive and process requests written in the syntax of ABDL.

Related to the goal of avoiding modifications to the functionality of MDBS is the goal of keeping the syntax of ABDL intact. We utilize existing ABDL constructs in our query translations. A single SQL query may map to one ABDL request or a series of ABDL requests. The processing of one

request may depend on the results of some other request in the series. Clearly, the interface must include some method of controlling the iterative processing of series of requests. The software structure of our proposed interface (described in Chapter VI and augmented in Chapter VII) provides for this iterative control.

As discussed above, we have made it our goal to leave MDBS and ABDL unchanged. We also desire to make our interface transparent to the SQL user. For example, in a corporate environment, a new employee with previous experience with SQL/DS should be able to log in at a system terminal, input a SQL request, and receive result data in a relational format (i.e., a table). The employee requires no training in MDBS or ABDL procedures prior to utilizing the system. An obvious advantage is that the new employee becomes a contributing member of the organization almost immediately, with no retraining. The non-productive period of new employee indoctrination is greatly reduced.

B. APPROACH TO DESIGN

Our approach to the design and analysis of a SQL interface to MDBS is unconventional by today's standards. The normal method is to design a system in a top-down manner. High-level abstractions are considered first, while deferring lower-level details. In this thesis, we consider the lowest levels first. We are building upon the basic subset of SQL-to-ABDL mappings provided by Macy [Ref. 8].

As additional SQL operations are incorporated into the interface, we make appropriate additions to the set of SQL-to-ABDL mappings. The functional requirements of an overall software structure for the interface become apparent in Chapter V, when we present ABDL translations for the nested SQL SELECT. The functionality and organization of structure components is described graphically, in text, and through the presentation of high-level algorithms. We reiterate that, in the development of the SQL interface, MDBS is considered to be a "black box" which processes database requests presented in the syntax of ABDL. We are proposing an interface, residing on a host computer, which enables a user to access a relational database implemented on an attribute-based system. Recommendations for modification within the structure of MDBS are made only if a desirable SQL operation cannot be supported by existing ABDL operations.

Our approach to the presentation of SQL-to-ABDL mappings is as follows. We first review the direct mappings (i.e., SELECT/RETRIEVE, INSERT/INSERT, DELETE/DELETE, and UPDATE/UPDATE) developed by Macy [Ref. 8]. Beginning in Chapter IV, we investigate additional operations to be supported by the interface. The functionality of each of these operations is thoroughly explained through the use of example queries. The equivalent ABDL requests are then determined.

All examples on database operations presented in this thesis are based on the Suppliers-and-Parts database depicted in Date [Ref. 9]. This database contains three relations: "S" (Suppliers), "SP" (Shipments), and "P" (Parts). We use many of Date's examples directly because they are well-known, thereby facilitating reader understanding of our SQL to ABDL translations. The database is depicted in Figure 1.

C. ORGANIZATION OF THE THESIS

In Chapter II, we present an overview of the organization and functionality of the Multi-backend Database System (MDBS). Also presented are descriptions of the attribute-based data language (ABDL) and the relational data language (SQL). Chapter III reviews the direct SQL-to-ABDL mappings as developed by Macy [Ref. 8]. SQL set membership operations involving single relations, and the equivalent ABDL requests are explained in Chapter IV. Chapter V explains set membership operations on multiple relations (i.e., nested SELECT). In Chapter VI, a software structure is proposed to facilitate the implementation of nested SELECTs. In Chapter VII, the interface software structure is modified to include the functionality necessary to accomplish the translation of other single-relation and multiple-relation operations. Chapter VIII presents our conclusions and recommendations for future research.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

Figure 1. The Suppliers-and-Parts Database.

II. THE MULTI-BACKEND DATABASE SYSTEM (MDBS), ITS DATA LANGUAGE (ABDL) AND THE INTERFACE LANGUAGE (SQL)

As we begin our investigation into the feasibility of designing and implementing a complete relational interface for the Multi-backend Database System (MDBS), it is important to gain a general familiarity with the organization of MDBS and with the system's attribute-based data language (ABDL). We have selected the Structured Query Language (SQL) as the relational data language to be supported by our interface. Therefore, we must also have an understanding of the structure and capabilities of this language.

In Sections A and B, we briefly describe MDBS and ABDL, respectively. Section C provides a brief description of SQL. These descriptions, though somewhat superficial, should enable the reader to comfortably follow subsequent discussions. A complete description of MDBS and ABDL can be found in Hsiao [Refs. 4 and 10]. The reader is referred to Astrahan [Ref. 11] and Chamberlin [Ref. 12] for in-depth discussions of SQL.

A. A REVIEW OF THE MULTI-BACKEND DATABASE SYSTEM (MDBS)

MDBS is a multiple-minicomputer backend database computer. Off-the-shelf hardware and specialized software are combined to provide database management service to a

host computer. Figure 2 depicts the hardware organization of MDBS. The hardware organization includes one minicomputer as a controller and multiple minicomputers as backends. Each backend has one or more dedicated disk drives. The controller and the backends are connected by a broadcast bus. The database is distributed across the disk drives of the backend in such a manner that the backends can process requests in parallel, providing a significant performance advantage over traditional single-processor architectures.

The prototype MDBS, currently operating at the U.S. Naval Postgraduate School, uses a VAX 11/780 as the controller and two PDP 11/44s as the backends. Each of these backends has one or more disk drives for its dedicated use. The multiple backends and the controller are connected by DEC's Parallel Communication Links (PCLs). Their broadcast capabilities are simulated in software.

The major design goal of MDBS is to provide a high-performance system for large-capacity databases. Throughput improvement should be proportional to the number of backends, and the response-time reduction should be inversely proportional to the number of backends. A second design goal is that the system should be easily extensible. The system should be able to accomodate additional backends with no modification to existing software, and no new programming. The incorporation of additional backends

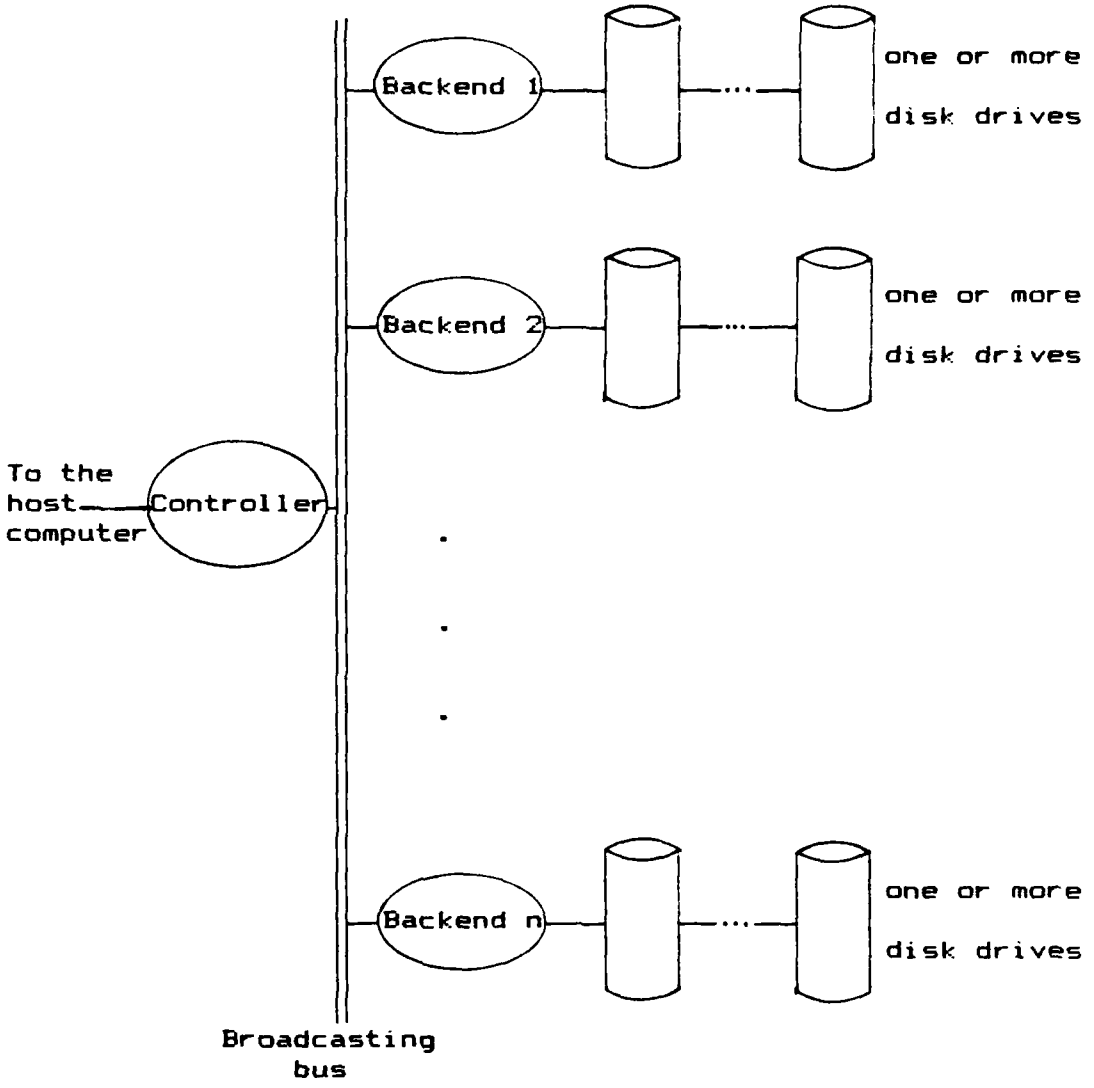


Figure 2. The MDBS Hardware Organization

should not require modification to existing hardware, and disruption of system activity should be minimal. The software structure of MDBS provides this extensibility. The software of the backends is identical, utilizing identical operating software for the additional backends.

It is clear that the controller could become a bottleneck. MDBS reduces this potential by minimizing the role of the controller and maximizing the amount of work done by the backends. The software structure of MDBS is shown in Figure 3. The functions of the controller are limited to request preparation, insert information generation, and post processing. The request preparation functions are performed before a request is placed on the broadcast bus. These functions handle parsing, syntax checking, and the transformation of a parsed request into the form required for processing at the backends. The insert information generation functions are performed during the processing of an insert request. These functions provide additional information to the backends, such as the identity of the particular backend at which the record is to be inserted. The post processing functions are performed after replies are returned from the backends. For example, result data are collected prior to forwarding to the host computer.

As described above, the controller does relatively little work. The backends, on the other hand, are

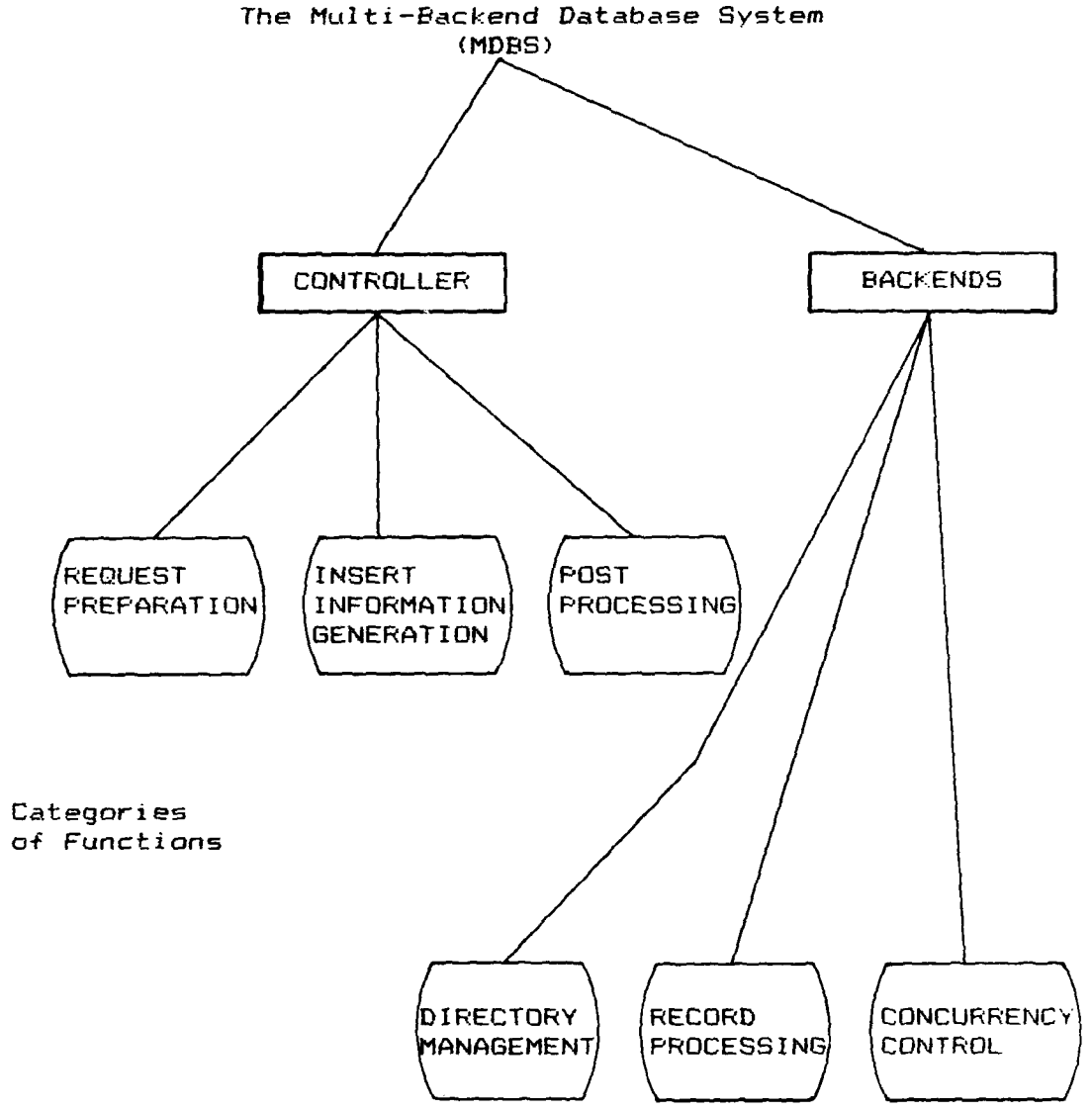


Figure 3. The MDBS Software Structure

responsible for all the major database management functions. These are directory management, record processing, and concurrency control. The directory management functions determine the secondary storage addresses of the appropriate records and perform directory table maintenance. The record processing functions store records into secondary storage, retrieve records from secondary storage, and select the records that contain the desired information. The concurrency control functions ensure consistency for concurrent execution of user requests.

The key to high-performance is in the parallelism of the backends. The database is distributed across the disks of all of the backends. Therefore, when a request is broadcasted from the controller, each backend can execute the request on its portion of the database. To yield an additional performance advantage, a queue of requests is maintained at each backend. Each backend schedules requests for execution independent of the activities of the other backends.

B. THE ATTRIBUTE-BASED DATA LANGUAGE (ABDL)

We preface our discussion of the syntax and functionality of ABDL with a brief introduction to the data model supported by MDBS. This model is the attribute-based data model, originally developed by Hsiao [Ref. 2]. The following constructs are informally defined. A database consists of a collection of files. Each file contains a

unique group of records. Each record is composed of two parts. The first of these parts is a collection of attribute-value pairs or keywords. An attribute-value pair is an element of the Cartesian product of the attribute name and the domain of attribute values. As an example, <STATUS,30> is an attribute-value pair having 30 as the value for the STATUS attribute. In each record, there is at most one attribute-value pair for each distinct attribute defined in the database. The last part of each record contains textual information. This is the record body. An example of a record without a record body is shown below. We note that all examples in this and subsequent sections are based on Date's suppliers-and-parts database as described in [Ref 9] and in Chapter I.

(<FILE,S>,<S#,S1>,<SNAME,Smith>,<STATUS,20>,<CITY,London>)

The first attribute-value pair in every record indicates the file name. In the example above, the file name is 'S' (the Suppliers file).

The database can be accessed through the use of keyword predicates. Each of these keyword predicates is a three-tuple of the form (attribute, relational_operator, value), e.g., (STATUS < 30). When keyword predicates are combined into a conjunction such as

$$((\text{FILE} = \text{S}) \wedge (\text{STATUS} < 30))$$

or into a disjunction of conjunctions such as

$$(((\text{FILE} = \text{S}) \wedge (\text{SNAME} = \text{Smith})) \vee$$
$$((\text{FILE} = \text{S}) \wedge (\text{SNAME} = \text{Jones})))$$

a query (in disjunctive normal form) of the database is formed.

In the following subsections, we will see how these keyword predicates and queries are used in the attribute-based data language for search and retrieval operations. We describe the syntax and functionality of the four types of request supported by ABDL: retrieve, insert, delete, and update. Appendix A provides a formal specification of this non-procedural language.

1. The RETRIEVE Request

The RETRIEVE request allows the user to query the database for information. This operation obtains the requested data without altering the database. The syntax is:

RETRIEVE (Query) <Target-list> [BY attribute] [WITH Pointer]

The type of the request is indicated by the reserved word RETRIEVE. As we have seen, the Query part is composed of predicates in the disjunctive normal form. From our previous discussion, we note that the Query specifies the file and those records within the file which satisfy the request. The attributes for which values are to be

extracted from this portion of the database are contained in the Target-list. ABDL supports five aggregate operations: AVG, COUNT, MAX, MIN, and SUM. Therefore, the attribute value may be an aggregate of values from multiple records, or the value from a single record.

The BY and WITH clauses are optional, as indicated by the square brackets in the syntax. The BY-clause is used when a grouping by some attribute is desired. The WITH-clause specifies whether pointers to the retrieved records must be returned to the user for later use in an update request. As an example of a RETRIEVE request, if we wish to obtain supplier names for all of the suppliers with STATUS greater than 10, grouped by location, we may use the following query:

RETRIEVE ((FILE = S) ∧ (STATUS > 10)) <SNAME> BY CITY

2. The INSERT Request

The INSERT request alters the database by adding a new record. The syntax is:

INSERT Record

An example of an INSERT request is:

INSERT (<FILE,S>, <S#,S1>, <SNAME,Smith>)

This adds a record to the suppliers file for supplier number S1 and identifies that supplier as Smith.

3. The DELETE Request

The DELETE request alters the database by removing an existing record or records. The syntax is:

DELETE Query

where Query specifies which records are to be deleted. An example of a DELETE request is:

DELETE ((FILE = S) ∧ (STATUS = 10))

This deletes all records in the suppliers file for suppliers whose status is equal to 10.

4. The UPDATE Request

The UPDATE request alters the database by modifying the value of some attribute in an existing record. The syntax is:

UPDATE Query Modifier

where Modifier indicates which of five types of modification is to be performed. These modifiers are defined as follows. A type-0 modifier sets the new value of the attribute being modified to a constant. A type-I modifier sets the new value of the attribute to be some function of its old value in the record being modified. A type-II modifier sets the new value to be some function of another attribute value in the record being modified. A type-III modifier sets the new value to be some function of another attribute value in another record identified by the Query in the modifier. A

type IV modifier sets the new value to be some function of another attribute value in another record identified by the pointer in the modifier. An example of an UPDATE request (using a type-I modifier) is:

UPDATE (FILE = S) <STATUS = STATUS + 10>

which adds 10 to the status of all suppliers.

C. THE RELATIONAL QUERY LANGUAGE (SQL) AS THE INTERFACE LANGUAGE

AS indicated in Chapter I, we have selected the Structured Query Language (SQL) as the data language to be supported by our relational interface to the Multi-backend Database System (MDBS). The language's commercial availability coupled with its simple yet powerful functionality make SQL an ideal choice.

In the preceding section, we described the attribute-based data model prior to introducing ABDL. However, in this section, we assume a certain familiarity with the relational data model as we prepare to describe the four basic constructs of SQL: SELECT, INSERT, DELETE, and UPDATE. If the reader desires a review of relational theory, there are several very good texts available. In particular, we recommend Date [Ref. 9] and Ullman [Ref. 13]. A discussion of the mapping between the relational data model and the attribute-based data model can be found in Banerjee [Ref. 6].

1. The SELECT Query

Data retrieval, which is represented syntactically as a SELECT-FROM-WHERE block, is the most basic operation of SQL. Mapping indicates that a known quantity (STATUS = 30) is to be transformed into a desired quantity (SNAME) by means of a relation (S). The attributes to be returned are listed in the SELECT clause (the built-in functions COUNT, SUM, AVG, MAX, and MIN may be applied to these attributes). The FROM clause indicates which relation or relations are to be searched. The WHERE clause specifies the retrieval conditions. As an example, if we desire to obtain the names of suppliers whose status is 30, we may use the following query:

```
SELECT   SNAME
FROM     S
WHERE    STATUS = 30
```

The SELECT construct allows the user great flexibility in data retrieval operations. The user can list several relations in the FROM clause in order to obtain values selected from more than one relation (JOIN operations). The WHERE clause can contain any number of predicates including the six standard relational operators (=, ^=, >, >=, <, and <=), and the Boolean operators (AND, OR, and NOT). Parenthesis may be used to indicate a desired order of evaluation. The set comparison operators IN, ANY,

and ALL may also be used in the WHERE clause. (We investigate the use of these operators in Chapter IV.)

There are many other possible variations to the SELECT operation including the extremely useful nested SELECT. In the nested SELECT, the result of one SELECT request is used in the WHERE clause of another SELECT request. (The nested SELECT is thoroughly described in chapter V.)

2. The INSERT Query

The INSERT request allows the user to insert a new tuple (row) or set of tuples into an existing relation (table). Insertion of a single tuple can be accomplished through the use of a query such as

```
INSERT INTO S:
        <'S6','Rollins','40','Newport'>
```

In this example, all of the attributes are present and in the correct order. If some attribute values are unknown, those attributes for which values are being inserted must be listed following the relation name. A SQL INSERT statement may also evaluate a SELECT request and insert the resulting set of tuples into an existing (or temporary) relation. An example of such an INSERT operation is as follows.

```
INSERT INTO TEMP:
        SELECT    P#
        FROM      SP
        WHERE     S# = 'S2'
```

This enters into TEMP part numbers for all parts supplied by supplier S2.

3. The DELETE Query

The DELETE specifies tuples to be removed from the database. The tuples are indicated by means of a WHERE clause that is syntactically identical to the WHERE clause of a SELECT construct. As an example, to delete supplier number five from the supplier relation, we may use the following query.

```
DELETE    S
WHERE     S# = 'S5'
```

We may also delete all shipments with the query

```
DELETE    SP
```

The SP relation is still known, but it is now empty.

4. The UPDATE Query

The UPDATE request is syntactically similar to the DELETE request, except that a SET clause is used to specify the updates to be made to the selected tuples. New attribute values contained in the SET clause may be stated as constants, as expressions based on the original value of

the attribute, or as nested queries. An example of an UPDATE request is

```
UPDATE  S
SET     STATUS = 2 * STATUS
WHERE   CITY = 'London'
```

This doubles the status of all suppliers in London.

III. REVIEW OF BASIC MAPPINGS

As we have described in Chapter II, the four primary database operations of the Structured Query Language (SQL) are SELECT, INSERT, DELETE, and UPDATE. Macy [Ref. 8] has shown that for a subset of simple, single-relation SQL queries of all four types, there exist direct mappings into requests of the Attribute-based Data Language (ABDL). These mappings are fundamental to all further SQL-to-ABDL translations introduced in this thesis. Therefore, in the remainder of this chapter, we provide a review of these basic mappings as defined by Macy. We explain the mappings both graphically and in text. Each graphical presentation will display the general forms of the SQL and ABDL constructs, and the mappings between them (such as Figure 4, which depicts the SELECT to RETRIEVE mapping). Sample translations, utilizing our suppliers-and-parts database, will be presented in the text. The subset of SQL, for which translations are described, contains those operations that Macy has determined can be directly supported by MDBS and ABDL. In the next chapter, we will show that SELECT requests involving set comparison operators can also be directly supported. In subsequent chapters, we describe translations for SQL constructs such as the nested SELECT which involve multiple ABDL constructs.

Prior to describing the specific SQL to ABDL mappings (e.g., SELECT to RETRIEVE), we discuss two general types of mapping identified by Macy: Syntactic-substitution mapping and Conversion mapping. Syntactic-substitution mappings are accomplished by simple substitution of syntactical terms. Mappings requiring only substitution are denoted by a directional arrow labeled with a square containing the letter S (e.g., the mapping between the reserved words SELECT and RETRIEVE in Figure 4). Conversion mappings are accomplished by combining a clause from an SQL query with information about the ABDL data structure to create the equivalent clause of the ABDL construct. Mappings requiring conversion are denoted by a directional arrow labeled with a triangle containing the letter C (e.g., the mapping between the SQL FROM and WHERE clauses to the ABDL Query in Figure 4). We will describe conversion mappings in more detail as we present each for the SQL to ABDL translations. For an extensive discussion of the basic mappings described in this chapter, the reader is referred to Macy [Ref. 8].

A. MAPPING THE SQL SELECT QUERY TO THE ABDL RETRIEVE REQUEST

The mapping from the SQL SELECT to the ABDL RETRIEVE is depicted in Figure 4. The mapping proceeds as follows. The reserved word SELECT is mapped by syntactic substitution to the reserved word RETRIEVE. The sel_expr_list maps directly to the target_list. A conversion mapping is

required to translate the FROM and WHERE clauses to the ABDL query clause. This is accomplished by creating an equality keyword-predicate for the relation_name, e.g., FILE = relation_name. This new predicate is combined with the

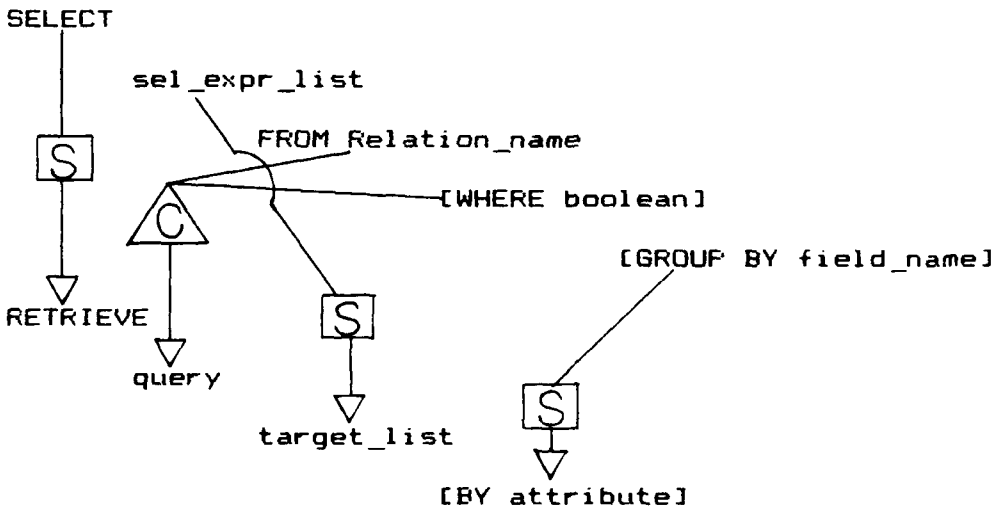


Figure 4. Mapping the SQL SELECT to the ABDL RETRIEVE

other predicates listed in the boolean expression to form an equivalent ABDL query clause. This conversion is called a query-conversion mapping. The GROUP BY construct maps directly to the BY construct. As an example of a SELECT to RETRIEVE translation, the following SQL SELECT will, for each part supplied, get the part number and the total quantity supplied of that part.

```
SELECT    P#,SUM(QTY)
FROM      SP
GROUP BY  P#
```

An equivalent ABDL request is

```
RETRIEVE (FILE = SP) <P#,SUM(QTY)> BY P#
```

B. MAPPING THE SQL INSERT QUERY TO THE ABDL INSERT REQUEST

The mapping from the SQL INSERT to the ABDL INSERT is depicted in Figure 5. The mapping proceeds as follows. The reserved word INSERT is the same for both requests. A conversion mapping, referred to as a record-conversion mapping, in this case, is required to translate "INTO relation_name insert_spec" into the ABDL "record". As we have seen in Chapter II, the ABDL record is a series of attribute-value pairs, the first pair of which identifies the file name. This mapping, then, can be accomplished by

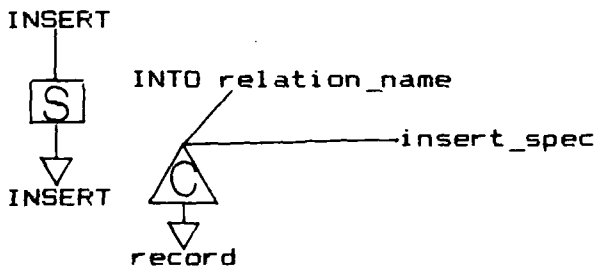


Figure 5. Mapping the SQL INSERT to the ABDL INSERT

constructing attribute-value pairs for the relation/file and relation/file_name and for the values of the attributes

listed in the insert_spec. As an example of an SQL INSERT to ABDL INSERT translation, the following SQL INSERT query will add part P7 (name 'Washer', color 'Grey', weight '2', city 'Athens') to relation/file P.

```
INSERT INTO P:
        <'P7','Washer','Grey','2','Athens'>
```

An equivalent ABDL request is

```
INSERT (<FILE,P>,<P#,P7>,<PNAME,Washer>,
        <COLOR,Grey>,<WEIGHT,2>,<CITY,Athens>)
```

C. MAPPING THE SQL DELETE QUERY TO THE ABDL DELETE REQUEST

The mapping from the SQL DELETE to the ABDL DELETE is depicted in Figure 6 The mapping proceeds as follows. The reserved word DELETE is the same for both requests. The query-conversion mapping, as described in Section A,

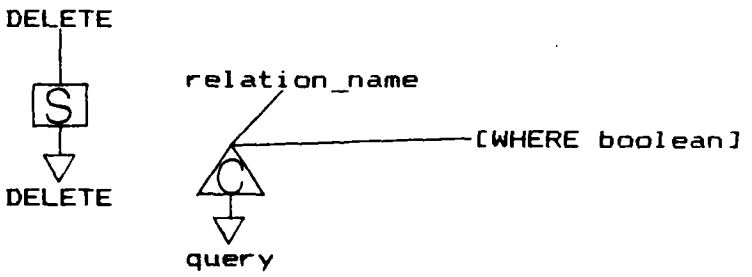


Figure 6. Mapping the SQL DELETE to the ABDL DELETE

is used to translate "relation_name" and "WHERE boolean" into the ABDL query clause. As an example of an SQL DELETE

to ABDL DELETE translation, the following SQL DELETE query will delete supplier S1 from the suppliers relation.

```
DELETE   S
WHERE    S# ='S1'
```

An equivalent ABDL request is

```
DELETE ((FILE = S) ∧ (S# = S1))
```

D. MAPPING THE SQL UPDATE QUERY TO THE ABDL UPDATE REQUEST

The mapping from the SQL UPDATE to the ABDL UPDATE is depicted in Figure 7. The mapping proceeds as follows.

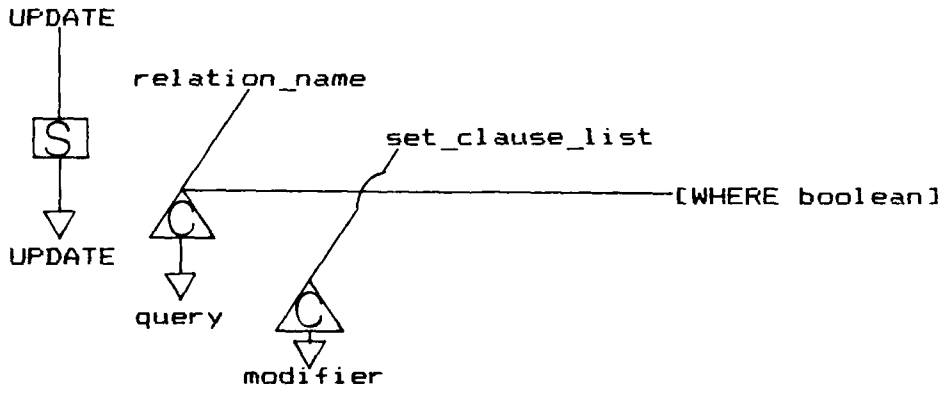


Figure 7. Mapping the SQL UPDATE to the ABDL UPDATE

The reserved word UPDATE is the same in both requests. As in Sections A and C, the query-conversion mapping is used to translate "relation_name" and "WHERE boolean" into the ABDL query clause. This conversion is common to the SELECT/RETRIEVE, DELETE, and UPDATE translations. The

component "set_clause_list" directly correlates to the ABDL "modifier", i.e., both constructs specify how the records being modified are to be updated. To accomplish this translation, the modifier conversion mapping is used. The conversion required is a restructuring of SQL set_clause_list constructs into acceptable ABDL format. The modifier-conversion is similar to the query-conversion. We now present an example of the conversions that are required in the translation of an SQL UPDATE to an ABDL UPDATE. If we desire to double the status of all supplierS in london, we may use the following SQL query:

```
UPDATE  S
SET     STATUS = 2 * STATUS
WHERE   CITY = 'London'
```

An equivalent ABDL request is

```
UPDATE ((FILE = S) ∧ (CITY = London)) (STATUS = 2 * STATUS)
```

IV. SELECTIONS WITH SET MEMBERSHIP OPERATIONS ON SINGLE RELATIONS

As we have seen, the condition following the WHERE clause in SQL SELECT operations may include the normal comparison operators, i.e., =, ^=, etc. Macy [Ref. 8] has shown that MDBS supports simple, single-relation retrieval operations using these comparison operators. SQL allows the use of several additional comparison operators. Three of these, IN, ANY, and ALL, deal with the set membership, and are of particular interest to us as we investigate possible extensions to the subset of SQL operations whose interfaces were proposed by Macy.

In this chapter we show how qualifications using IN, ANY, and ALL can be supported by MDBS. We first consider the simple case where set members are enumerated in the query. Some of the examples we provide herein may not appear very useful. However, they will serve to illustrate the mechanics of SELECT operations using these comparison operators. Their usefulness will become apparent in Chapter V, when we use them in retrievals involving multiple levels of nesting.

In sections A, B, and C, we formally define the comparison operators IN, ANY, and ALL, respectively. As noted by Chamberlin, et. al. [Ref. 14], English language definitions of these operators are, at best, ambiguous. We

shall, nevertheless, attempt to explain them in text prior to providing a clarifying definition in predicate logic. An example of a SELECT query will then be given for each case. The result relation of each of these examples will be provided in order to further clarify the uses of these operators. As in previous chapters, our examples specify retrievals of data contained in Date's database (defined in Chapter I). We will continue to utilize this database throughout this thesis. Again, note that some of our examples are taken directly from Date [Ref. 9]. In Sections D, E, and F we express IN, ANY, and ALL in the ABDL requests.

A. IN-MEMBERSHIP OPERATIONS

The comparison operator, IN, can be thought of as the set membership operator, $\in$. Correspondingly, NOT IN is equivalent to $\notin$.

1. The Set Membership Operator, 'IN'

The operator, IN, is evaluated as follows. The condition, A IN B, evaluates to be true if and only if the value of attribute A is equal to at least one value in the enumerated set B. The formal definition in predicate logic follows:

$$\forall x \ (x \in A \iff \exists y \ (y \in B \mid x = y))$$

EXAMPLE 1: If we wish to obtain supplier numbers for suppliers Smith and Jones, we may use the following query:

```
SELECT    S#,SNAME
FROM      S
WHERE     SNAME IN (Smith,Jones)
```

The result relation is:

S#	SNAME
S1	Smith
S2	Jones

2. The Set Membership Operator, 'NOT IN'

The operator, NOT_IN, is evaluated as follows. The condition, A NOT_IN B, evaluates to be true if and only if the value of attribute A is not equal to any value in the enumerated set B. The formal definition in predicate logic follows:

$$\forall x \ (x \in A \Longleftrightarrow \forall y \ (y \in B \mid x \neq y))$$

EXAMPLE 2: If we wish to obtain supplier numbers for suppliers who supply some parts, but do not supply parts P3 or P4, we may use the following query:

```
SELECT    S#
FROM      SP
WHERE     P# NOT IN (P3,P4)
```

The result relation is:

S#
S2
S3

B. ANY-MEMBERSHIP OPERATIONS

The comparison operator, ANY, is used in conjunction with the six standard relational operators, =, ~=, <=, >=, <, and >. It specifies variations on the theme of set membership as explained in the following subsections.

1. The Set Membership Operator, '=ANY'

The operator, =ANY, is interchangeable with the operator, IN. The condition, A =ANY B, evaluates to be true if and only if the value of attribute A is equal to at least one value in the enumerated set B. Example 1 and the predicate logic definition given for the operator IN apply equally to =ANY. In subsequent examples involving set membership, we shall use IN rather than =ANY.

2. The Set Membership Operator, '~=ANY'

The operator, ~=ANY, is evaluated as follows. The condition, A ~=ANY B, evaluates to be true if and only if the value of attribute A is not equal to at least one value in the enumerated set B. The formal definition in predicate logic follows:

$$\forall x \; (x \in A \iff \exists y \; (y \in B \mid x \sim= y))$$

EXAMPLE 3: If we wish to obtain supplier numbers for suppliers who supply some parts, but do not

supply both parts P1 and P2, we may use the following query:

```
SELECT    S#
FROM      SP
WHERE     P# ~=ANY (P1,P2)
```

The result relation is:

S#
S3
S4

3. The Set Membership Operator, '<=ANY'

The operator, <=ANY, is evaluated as follows. The condition, A <=ANY B, evaluates to be true if and only if the value of attribute A is less than or equal to at least one value in the enumerated set B. This implies that the value of attribute A is less than or equal to the maximum value in the set B. <=ANY, then, is not particularly useful in the case of enumerated sets. The operators >=, >, and < are similarily of limited value when sets are enumerated in the query. As previously stated, the usefulness of these operators will become apparent when we discuss queries in which the results of one SELECT operation determine the set members in the WHERE clause of another SELECT operation (nested SELECT). The formal predicate logic definition of A <=ANY B follows:

$$\forall x \ (x \in A \iff \exists y \ (y \in B \mid x <= y)) \implies$$
$$\forall x \ (x \in A \iff x <= \max \{B\})$$

As can be seen from the predicate logic definition, when using the operator, <=ANY, it is logically unnecessary to list more than one value (the maximum value) in the enumerated set B. A similar comment is applicable when using >=ANY, <ANY, or >ANY. However, in anticipation of our nested SELECT discussion in Chapter V, example queries utilizing these operators will each contain an enumerated set having more than one member. The additional values listed in the set are superfluous. However, they will help demonstrate the differing results obtained through the use of the ANY and ALL operators.

EXAMPLE 4: If we wish to obtain supplier names for suppliers whose status is not larger than 30, we may use the following query:

```
SELECT   SNAME
FROM     S
WHERE    STATUS <=ANY (10,20,30)
```

The result relation is:

SNAME
Smith
Jones
Blake
Clark
Adams

4. The Set Membership Operator, '>=ANY'

The operator >=ANY is evaluated as follows. The condition A >=ANY B evaluates to true if and only if the

value of attribute A is greater than or equal to at least one value in the enumerated set B. This implies that the value of attribute A is greater than or equal to the minimum value in the set B. The formal definition in predicate logic follows:

$$\forall x \ (x \in A \ <==> \ \exists y \ (y \in B \mid x >= y)) \ ==>$$
$$\forall x \ (x \in A \ <==> \ x >= \min (B))$$

EXAMPLE 5: If we wish to get supplier names for suppliers whose status is not less than 10, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS >=ANY (10,20,30)
```

The result relation is:

SNAME
Smith
Jones
Blake
Clark
Adams

5. The Set Membership Operator, '<ANY'

The operator, <ANY, is evaluated as follows. The condition, A <ANY B, evaluates to be true if and only if the value of attribute A is less at least one value in the enumerated set B. This implies that the value of attribute A is less than the maximum value in set B. The formal predicate logic definition follows:

$$\forall_x\ (x \in A \Longleftrightarrow \exists_y\ (y \in B \mid x < y)) \Longrightarrow$$
$$\forall_x\ (x \in A \Longleftrightarrow x < \max\ (B))$$

EXAMPLE 6: If we wish to obtain supplier names for suppliers whose status is less than 30, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS <ANY (10,20,30)
```

The result relation is:

SNAME
Smith
Jones
Clark

6. The Set Membership Operator, '>ANY'

The operator, >ANY, is evaluated as follows. The condition, A >ANY B, evaluates to be true if and only if the value of attribute A is greater than at least one value in the enumerated set B. The formal predicate logic definition follows:

$$\forall_x\ (x \in A \Longleftrightarrow \exists_y\ (y \in B \mid x > y)) \Longrightarrow$$
$$\forall_x\ (x \in A \Longleftrightarrow x > \min\ (B))$$

EXAMPLE 7: If we wish to obtain supplier names for suppliers whose status is greater than 10, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS >ANY (10,20,30)
```

The result relation is:

SNAME
Smith
Blake
Clark
Adams

C. ALL-MEMBERSHIP OPERATIONS

Like the comparison operator, ANY, the operator, ALL, is used in conjunction with the six standard relational operators. It also specifies variations on the set membership theme.

1. The Set Membership operator, '=ALL'

The operator, =ALL, is evaluated as follows. The condition, A =ALL B, evaluates to be true if and only if the value of attribute A is equal to every (each) value in the enumerated set B. The formal predicate logic definition follows:

$$\forall x\ (x \in A \Longleftrightarrow \exists y\ (y \in B \mid x = y)) \wedge$$
$$\forall y\ (y \in B \Longleftrightarrow \exists x\ (x \in A \mid x = y))$$

From this definition, it is apparent that the set B, whether manually enumerated or determined by the results of an inner SELECT, would contain only one value (or duplicates of that value). Therefore, since we can always use a condition of

the form WHERE STATUS = 30, we shall not use the operator =ALL in further discussion or examples.

2. The Set Membership Operator, '~=ALL'

The operator, ~=ALL, is interchangeable with the operator, NOT_IN. The condition, A ~=ALL B, evaluates to be true if and only if the value of attribute A is not equal to every value in the enumerated set B. In other words, there is no value in the set B to which the value of attribute A is equal. The predicate logic definition of NOT_IN is repeated for clarity:

$$\forall x \ (x \in A \Leftrightarrow \forall y \ (y \in B \mid x \sim= y))$$

The query given in example 2 (with ~=ALL substituted for NOT IN) is applicable. In subsequent examples involving set membership, we shall use NOT IN rather than ~=ALL.

3. The Set Membership operator, '<=ALL'

The operator, <=ALL, is evaluated as follows. The condition, A <=ALL B, evaluates to be true if and only if the value of attribute A is less than or equal to every value in the enumerated set B. This implies that the value of attribute A is <= the minimum value in set B. The predicate logic definition follows:

$$\forall x \ (x \in A \Leftrightarrow \forall y \ (y \in B \mid x <= y)) \Rightarrow$$
$$\forall x \ (x \in A \Leftrightarrow x <= \min (B))$$

Again, as in the case of the operator ANY, our degenerate

examples utilizing the operators <=ALL, >=ALL, <ALL, and >ALL will be presented with enumerated sets containing more than one member (even though, logically, only one member is necessary).

EXAMPLE 8: If we wish to obtain supplier names for suppliers whose status is not greater than 10, we may use the following query:

```
SELECT   SNAME
FROM     S
WHERE    STATUS <=ALL (10,20,30)
```

The result relation is:

SNAME
Jones

Note that the difference between the comparison operators ANY and ALL is readily apparent when we compare this example with example 4. In example 4, the operator, <=ANY, allows us to obtain supplier names for suppliers whose status is not larger than 30. The result relation in that example includes the names of all five suppliers.

4. The Set Membership Operator, '>=ALL'

The operator, >=ALL, is evaluated as follows. The condition, A >=ALL B, evaluates to be true if and only if the value of attribute A is greater than or equal to every value in the enumerated set B. This implies that the value of attribute A is greater than or equal to the maximum value

in set B. The predicate logic definition follows:

$$\forall x\ (x \in A \Longleftrightarrow \forall y\ (y \in B \mid x \geq y)) \Longrightarrow$$
$$\forall x\ (x \in A \Longleftrightarrow x \geq \max(B))$$

EXAMPLE 9: If we wish to obtain supplier names for suppliers whose status is at least 30, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS >=ALL (10,20,30)
```

The result relation is:

SNAME
Blake
Adams

5. The Set Membership operator, '<ALL'

The operator, <ALL, is evaluated as follows. The condition, A <ALL B, evaluates to be true if and only if the value of attribute A is less than every value in the enumerated set B. The predicate logic definition follows:

$$\forall x\ (x \in A \Longleftrightarrow \forall y\ (y \in B \mid x < y)) \Longrightarrow$$
$$\forall x\ (x \in A \Longleftrightarrow x < \min(B))$$

EXAMPLE 10: if we wish to obtain supplier names for suppliers whose status is less than 10, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS <ALL (10,20,30)
```

The result relation is:

SNAME

Note that this is the empty relation. There are no suppliers whose status is less than 10.

6. The Set Membership Operator, '>ALL'

The operator, >ALL, is evaluated as follows. The condition, A >ALL B, evaluates to be true if and only if the value of attribute A is greater than every value in the enumerated set B. The predicate logic definition follows:

$$\forall x \ (x \in A \Leftrightarrow \forall y \ (y \in B \mid x > y)) \Rightarrow$$
$$\forall x \ (x \in a \Leftrightarrow x > \max(B))$$

EXAMPLE 11: If we wish to obtain supplier names for suppliers whose status is greater than 30, we may use the following query:

```
SELECT    SNAME
FROM      S
WHERE     STATUS >ALL (10,20,30)
```

The result relation is:

SNAME

As in example 10, this is the empty relation. There are no suppliers whose status is greater than 30.

D. EXPRESSING IN-MEMBERSHIP OPERATIONS IN ABDL

In this and the following two sections, we present ABDL translations for the examples given in sections A, B, and C. Each SQL example will be repeated, followed by the ABDL translation.

1. The Set Membership Operator, 'IN'

The SQL query presented as example 1 is

```
SELECT    S#,SNAME
FROM      S
WHERE     SNAME IN (Smith,Jones)
```

Our proposed SQL interface would provide the following ABDL translation:

```
RETRIEVE (((FILE = S) ∧  (SNAME = Smith)) ∨
          ((FILE = S) ∧  (SNAME = Jones))) <S#,SNAME>
```

One conjunction is created for each value in the enumerated set, containing an equality predicate. The ABDL request will have as many conjunctions as there are values in the set.

2. The Set Membership operator, 'NOT IN'

The SQL query presented as example 2 is

```
SELECT    S#
FROM      SP
WHERE     P# NOT IN (P3,P4)
```

The ABDL translation is

RETRIEVE ((FILE = SP) ∧ (P# ~= P3) ∧ (P# ~= P4) <S#>

One predicate of the form (attribute ~= value) is created for each value in the enumerated set. The ABDL request will contain a single conjunction, which is the logical AND of these predicates.

E. EXPRESSING ANY-MEMBERSHIP OPERATIONS IN ABDL

1. The Set Membership Operator, '=ANY'

As previously defined, =ANY is equivalent to IN and will not be included in our set of allowable SQL constructs.

2. The Set Membership Operator, '~=ANY'

The SQL query presented as example 3 is

```
SELECT   S#
FROM     SP
WHERE    P# ~=ANY (P1,P2)
```

The ABDL translation is

RETRIEVE (((FILE = SP) ∧ (P# ~= P1)) ∨
((FILE = SP) ∧ (P3 ~= P2))) <S#>

One conjunction is created for each value in the enumerated set, containing a predicate of the form (attribute ~= value).

3. The Set Membership Operator, '<=ANY'

The SQL query presented as example 4 is

```
SELECT    SNAME
FROM      S
WHERE     STATUS <=ANY (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE =S) ∧ (STATUS <= 30)) <SNAME>
```

One predicate of the form (attribute <= max_value) is created. The ABDL request will contain a single conjunction. Note that the SQL interface recognizes that the condition in the WHERE clause evaluates to true if and only if a supplier's status is less than or equal to at least one of the status values in the enumerated set (implying that that supplier's status is less than or equal to the maximum value in the set). Therefore, only the maximum value, 30, is utilized in the ABDL translation.

4. The Set Membership Operator, '>=ANY'

The SQL query presented as example 5 is

```
SELECT    SNAME
FROM      S
WHERE     STATUS >=ANY (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE = S) ∧ (STATUS >= 10)) <SNAME>
```

One predicate of the form (attribute >= min_value) is created. The ABDL request will contain a single

conjunction. As in the '<=ANY' case, only one value of the enumerated set in the WHERE clause is utilized in the ABDL translation. In this case, the minimum value, 10, is utilized.

5. The Set Membership Operator, '<ANY'

The SQL query presented as example 6 is

```
SELECT   SNAME
FROM     S
WHERE    STATUS <ANY (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE = S) ∧ (STATUS < 30)) <SNAME>
```

One predicate of the form (attribute < max_value) is created. The ABDL request will contain a single conjunction.

6. The Set Membership Operator, '>ANY'

The SQL query presented as example 7 is

```
SELECT   SNAME
FROM     S
WHERE    STATUS >ANY (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE =S) ∧ (STATUS > 10)) <SNAME>
```

One predicate of the form (attribute > min_value) is created. The ABDL request will contain a single conjunction.

F. EXPRESSING ALL-MEMBERSHIP OPERATIONS IN ABDL

1. The Set Membership Operator, '=ALL'

As previously defined, use of the operator, =ALL, is equivalent to using the standard equality operator, =. We will, therefore, not include it in our set of allowable SQL constructs.

2. The Set Membership Operator, '~=ALL'

As previously defined, ~=ALL is equivalent to NOT_IN and will not be included in our set of allowable SQL constructs.

3. The Set Membership Operator, '<=ALL'

The SQL query presented as example 8 is

```
SELECT   SNAME
FROM     S
WHERE    STATUS <=ALL (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE = S) ∧ (STATUS <= 10)) <SNAME>
```

One predicate of the form (attribute <= min_value) is created. The ABDL request will contain a single conjunction. As in the '<=ANY' case, the translator in our SQL interface utilizes only one value from the enumerated

set. Note that in this case, the minimum value, 10, is chosen, whereas, in the '<=ANY' case the maximum value, 30, is chosen.

4. The Set Membership Operator, '>=ALL'

The SQL query presented as example 9 is

```
SELECT    SNAME
FROM      S
WHERE     STATUS >=ALL (10,20,30)
```

The ABDL translation is

```
RETRIEVE ((FILE = S) ∧  (STATUS >= 30)) <SNAME>
```

One predicate of the form (attribute >= max_value) is created. The ABDL request will contain a single conjunction. As in the '>=ANY' case, only one value of the enumerated set is utilized. In this case, the maximum value, 30, is utilized in the equivalent RETRIEVE construct. We recall that the minimum value, 10, was utilized in the '>=ANY' case.

5. The Set Membership Operator, '<ALL'

The SQL query presented as example 10 is

```
SELECT    SNAME
FROM      S
WHERE     STATUS <ALL (10,20,30)
```

The ABDL translation is

RETRIEVE ((FILE = S) / (STATUS < 10)) <SNAME>

One predicate of the form (attribute < min_value) is created. The ABDL request will contain a single conjunction.

6. The Set Membership Operator, '>ALL'

The SQL query presented as example 11 is

```
SELECT   SNAME
FROM     S
WHERE    STATUS >ALL (10,20,30)
```

The ABDL translation is

RETRIEVE ((FILE = S) ∧ (STATUS > 30)) <SNAME>

One predicate of the form (attribute > max_value) is created. The ABDL request will contain a single conjunction.

V. SELECTIONS WITH SET MEMBERSHIP OPERATIONS ON MULTIPLE RELATIONS

In the preceding chapter, we have described SQL SELECT queries which utilize the comparison operators, IN, ANY, and ALL in the WHERE clause. These are simple, single-relation queries in which the associated sets are enumerated. We now discuss the nested SQL SELECT queries (or nested mapping) in which the result of one mapping is used in the WHERE clause of another mapping. In other words, the membership of the set following IN, ANY, or ALL in one SELECT operation is determined by the result set of another SELECT. We will describe the operation of two-level, three-level and n-level nested SELECTs in Sections A, B, and C, respectively. In Section D, we show how the nested SQL SELECT is translated into a series of ABDL RETRIEVEs.

A. NESTED SELECTIONS WITH TWO RELATIONS

As previously stated, in a nested SQL SELECT, the results of one SELECT operation are used in the WHERE clause of another SELECT operation. We view the former SELECT as the inner (level of) SELECT, and the latter as the outer (level of) SELECT. Figure 8 depicts an example of a two-level nested SELECT operation. This particular example is chosen for its similarity to one of our examples in Chapter IV (i.e., Example 6) which utilizes the operator, <ANY, in

conjunction with a manually enumerated set. In the degenerate case presented in that example, the operator, <ANY, appeared to be of marginal usefulness. The usefulness of this and similar operators (e.g., <=ANY, >=ALL) in the nested SELECT, will now become apparent.

Both our current example in Figure 8, and Example 6 of Chapter IV result in a set of supplier numbers for suppliers with status value less than the current maximum status value in the S table. In our degenerate example, we must know (i.e., enumerate) that that value is 30. In our present example, we allow an inner SELECT to obtain the status value for each supplier number in the S table. By employing an inner level of SELECT, we are free from enumerating the values.

```
Outer
SELECT   SELECT   S#
         FROM     S
         WHERE    STATUS <ANY
Inner              (SELECT   STATUS
SELECT             FROM     S)
```

Figure 8. A Two-Level Nested SELECT

Processing of the two-level nested SELECT in Figure 8 proceeds as follows. First, the inner SELECT retrieves all status values in the S table. The result of this SELECT is the set (with duplicates) of status values {20,10,30,20,30}.

The outer SELECT then selects supplier numbers FROM table S WHERE the status value is less than at least one of the values in the above result set. The result relation is

S#
S1
S2
S4

B. NESTED SELECTIONS WITH THREE RELATIONS

We now describe a three-level nested SELECT. We present an example which demonstrates the usefulness of the set/comparison operator IN, and of multi-level SELECTs in general. In the course of providing the requested data, this three-level SELECT chooses data from each of the three tables which comprise our sample database. The request is to get supplier names for suppliers who supply at least one red part. The query is presented in Figure 9.

```
Outermost SELECT:   SELECT   SNAME
                    FROM     S
                    WHERE    S# IN
    Inner SELECT:       (SELECT   S#
                        FROM      SP
                        WHERE     P# IN
        Innermost SELECT:   (SELECT   P#
                            FROM      P
                            WHERE     COLOR = 'RED'))
```

Figure 9. A Three-Level Nested SELECT

Processing of the query in Figure 9 proceeds as follows.

Step 1: The innermost SELECT retrieves part numbers (P#) from the parts relation (P) where the color of the parts is red. The result of this SELECT is the set of part numbers {P1,P4,P6}.

Step 2: The next SELECT retrieves supplier numbers (S#) from the shipments relation (SP) where P#s are in the result set of step 1. The result of this SELECT is the set of supplier numbers {S1,S2,S4}.

Step 3: The outermost SELECT retrieves supplier names (SNAME) from the suppliers relation (S) where S#s are in the result set of step 2. The result relation passed to the user is

SNAME
Smith
Jones
Clark

C. NESTED SELECTIONS WITH N RELATIONS

Although it seems unlikely that many users would utilize a nested SELECT of more than 2 or 3 levels, the subqueries can be nested to any depth. The form of an n-level nested SELECT is shown in Figure 10.

The SET_OPR in Figure 10 refers to the various forms of our comparison operators IN, ANY, and ALL. In the next section, we describe the translation of nested SELECTs to a series of ABDL RETRIEVEs. Therefore, it is important that

```
SELECT sel_expr_list                                  )
                                                      )  level 1 or
FROM   relation_name_1                                )  outermost
                                                      )  SELECT
WHERE  attribute_name1 SET_OPR                        )

      (SELECT attribute_name1                         )
                                                      )  level 2 or
       FROM   relation_name_2                         )  inner
                                                      )  SELECT
       WHERE  attribute_name2 SET_OPR                 )
                    .
                    .
                    .
            (SELECT attribute_name(n-1)               )
                                                      )  level n or
             FROM   relation_name_n                   )  innermost
                                                      )  SELECT
             WHERE  condition)...)                    )
```

Figure 10. An N-Level Nested SELECT

we note the following information as succinctly stated in [Ref. 1].

> "The nth level is the innermost SELECT. The 1st level is the outermost SELECT. The sel_expr_list of each inner SELECT, i.e., a SELECT lower than level 1, contains a single attribute name, which is the same as the attribute name used in the qualification of the next-higher level SELECT. The relation names at any two levels may be the same."

D. TRANSLATING NESTED SELECTIONS TO A SERIES OF ABDL RETRIEVALS

As shown by Macy [Ref. 8], there exists a straightforward mapping between the SQL SELECT operation and the ABDL RETRIEVE operation. We can, therefore, simulate the nested SELECT with a series of RETRIEVEs, each succeeding operation using the results of the previous one. Thus, referring to our three-level example of Section B, the

ABDL equivalent of the innermost SELECT is

```
RETRIEVE ((FILE = P) ∧ (COLOR = 'RED')) <P#>
```

The resulting set of part numbers {P1,P4,P6} is then used in the next ABDL operation as follows:

```
RETRIEVE (((FILE = SP) ∧ (P# = P1)) ∨
          ((FILE = SP) ∧ (P# = P4)) ∨
          ((FILE = SP) ∧ (P# = P6))) <S#>
```

The last retrieve (corresponding to the outermost SELECT in our example) then uses the resulting set of supplier numbers {S1,S2,S4} as follows:

```
RETRIEVE (((FILE = S) ∧ (S# = S1)) ∨
          ((FILE = S) ∧ (S# = S2)) ∨
          ((FILE = S) ∧ (S# = S4))) <SNAME>
```

It is intended that the operation of our SQL interface be transparent to the SQL user. Therefore, the resulting values of the attribute SNAME (Smith,Jones,Clark) are returned to the user in the form of the result relation previously described for our three-level nested SELECT example of section B.

We have now demonstrated the operation of data retrievals involving the nested SELECT construct. These nested operations may include use of the various forms of IN, ANY, and ALL. The sequence of actions necessary to

translate the nested SQL SELECT to a series of ABDL RETRIEVEs has been described. In the next chapter, we present our proposals for the implementation of these translations.

VI. IMPLEMENTING NESTED SELECTIONS

The logical process by which a nested SQL SELECT is translated to a series of ABDL RETRIEVES has been described. It is clear that each SELECT level, from the innermost to the outermost, must be translated to an ABDL RETRIEVE. Then, each RETRIEVE is processed in turn, with each succeeding operation utilizing the results of the previous RETRIEVE in the QUERY part. In Section A of this chapter, we present the algorithms for building the ABDL QUERY. In Section B, a simple iterative structure for controlling the execution of n-level nested SELECTs is provided. Finally, in Section C, the overall software structure of our SQL interface will be proposed. Note that, as we continue our bottom-up investigation and include additional SQL operations in our set of allowable constructs, the functionality of this structure may be augmented. However, it is expected that the software structure will remain intact.

A. ALGORITHMS FOR BUILDING THE ABDL QUERY

We recall that the Query part of ABDL RETRIEVEs (DELETE and UPDATE, as well) is written in a disjunctive normal form. A QUERY may be a single conjunction or it may be a disjunction of conjunctions. The number of conjunctions generated in the translation of nested SELECTs utilizing the

various forms of IN, ANY, and ALL has been noted in Sections D, E, and F of Chapter IV. Figure 11 summarizes this information. The figure also specifies the relational operators involved, as well as the source of the values to be used in each conjunction.

Set Opr	# Conjunctions	Rel Opr	Value Source
IN	n	=	result set
NOT IN	1	~=	result set
~=ANY	n	~=	result set
<=ANY	1	<=	max(result set)
>=ANY	1	>=	min(result set)
<ANY	1	<	max(result set)
>ANY	1	>	min(result set)
<=ALL	1	<=	min(result set)
>=ALL	1	>=	max(result set)
<ALL	1	<	min(result set)
>ALL	1	>	max(result set)

Figure 11. Summary of Nested SELECT Set Comparison Operators

From Figure 11, it is clear that our translator must perform a multiway selection depending upon which set comparison operator is utilized at each SELECT level. We describe an appropriate algorithm in Subsection 1. It can also be seen that, in the case of the operators IN and

~=ANY, a number of conjunctions are generated, one for each value in the result set of the previous operation. In Subsection 2, we present an n-conjunction algorithm to handle these two cases. Note that in all remaining cases, a single conjunction is generated. The 1-conjunction algorithm is presented in Subsection 3.

1. The Query-Constructor Subroutine

As noted above, the top-level translator portion of our SQL interface must determine from the set comparison operator the proper algorithm for constructing the QUERY part of the resultant ABDL request. This can be handled by a multi-way selection or CASE construct, as shown in the Query-Constructor Algorithm in Figure 12. The parameters passed to Query_Constructor are Query_Template (a conjunction, described in Subsection 2, constructed to facilitate the incorporation of succeeding result sets), the Result_Set of the previous request, and the appropriate Set_Opr from Figure 11.

In each alternative of the CASE statement of Figure 12, the correct relational operator is chosen, and either the n-conjunction or the 1-conjunction subroutine is called. The parameters provided for each subroutine call are the relational operator and the result set of the previous operation, or the maximum/minimum value of the result set. As previously discussed, when ANY and ALL are used with these relational operators, only one value of the result set

```
Subroutine Query_Constructor(Query_Template,Result_Set,
                             Set_Opr)
  CASE Set_Opr OF

  IN:       Rel_Opr <-- '='
            call N_conjunction(Query_Template,Result_Set,
                               Rel_Opr);

  NOT IN:   Rel_Opr <-- '~='
            call One_conjunction(Query_Template,Result_Set,
                                 Rel_Opr);

  ~=ANY:    Rel_Opr <-- '~='
            call N_conjunction(Query_Template,Result_Set,
                               Rel_Opr);

  <=ANY:    Rel_Opr <-- '<='
            call One_conjunction(Query_Template,
                                 max(Result_Set),Rel_Opr);

  >=ANY:    Rel_Opr <-- '>='
            call One_conjunction(Query_Template,
                                 min(Result_Set),Rel_Opr);

  <ANY:     Rel_Opr <-- '<'
            call One_conjunction(Query_Template,
                                 max(Result_Set),Rel_Opr);

  >ANY:     Rel_Opr <-- '>'
            call One_conjunction(Query_Template,
                                 min(Result_Set),Rel_Opr);

  <=ALL:    Rel_Opr <-- '<='
            call One_conjunction(Query_Template,
                                 min(Result_Set),Rel_Opr);

  >=ALL:    Rel_Opr <-- '>='
            call One_conjunction(Query_Template,
                                 max(Result_Set),Rel_Opr);

  <ALL:     Rel_Opr <-- '<'
            call One_conjunction(Query_Template,
                                 min(Result_Set),Rel_Opr);

  >ALL:     Rel_Opr <-- '>'
            call One_conjunction(Query_Template,
                                 max(Result_Set),Rel_Opr);
  END CASE

END Query_Constructor
```

Figure 12. The Query_Constructor Subroutine

is utilized in the translation. Depending upon which form of the set comparison operator is used, the selected value will be either the maximum or the minimum value in the result set. Therefore, a call to a standard Max or Min function, as appropriate, must be made prior to sending the resultant single value to the 1-conjunction subroutine. It should be noted that the 1-conjunction subroutine is called in the case of the operator NOT IN. However, there is no need to utilize a Max/Min function. We also note that a call to Max/Min is never needed prior to a call to the n-conjunction subroutine.

2. The N-Conjunction Subroutine

In the case of the set operators IN and ~=ANY. the above Query-Constructor subroutine will call the n-conjunction subroutine. In the process of translating nested SELECTs which utilize these operators, one conjunction of the form

((FILE = Relname) ∧ (Attrname Rel_opr Value))

will be generated for each value in the result set. These conjunctions are ORed to form a disjunction of conjunctions, as explained in Chapter IV, Sections D and E. An algorithmic representation of the n-conjunction generation subroutine is provided in Figure 13.

The template, defined in Figure 13, is provided by the top-level translator as it translates each SELECT level

to an ABDL RETRIEVE. Value_of_Template is the only variable which requires substitution. For the innermost (nth level) SELECT of a nested SELECT request, the equivalent RETRIEVE can be constructed completely. However, at translation time, the values to be used in the query portion of the

```
Subroutine N_conjunction(Query_Template,Rel_opr)

   /* Query_Template:                                          */
   /*   is ((FILE = Relname) ∧  (Attrname Rel_opr Value))      */
   /* Query:                                                   */
   /*   is Query_Template  ∨ Query_Template  ∨ . . .           */
   /*                      ∨ Query_Template                    */
   /*                                                          */
   /* For every value in the Result_set                        */
   /* generate one conjunction using Template                  */
   /* then OR-concatenate into Query.                          */

  Rel_opr_of_Template <-- Rel_opr

  if Result_set is NOT EMPTY
    then
      Value_of_Template <-- 1st value from Result_set
      Query <-- Query_Template  /* Relname & Attrname          */
                                /* filled in                   */
        while more values in Result_set do
          Value_of_Template <-- next value from Result_set
          Query <-- Query || ' V' || Template
        end while
     else
       Query <-- ' '   /* Query is nil */

END N_conjunction
```

Figure 13. The N-conjunction Subroutine

remaining n-1 SELECTS are unknown. Therefore, the template is provided to the N-conjunction generator which fills in the missing values and constructs the QUERY part of each RETRIEVE.

3. The 1-Conjunction Subroutine

In the case of the operator NOT IN and all of the ANY/ALL operators containing <=, >=, <, or >, the CASE statement causes a call to the 1-conjunction subroutine. As described in Chapter IV, one predicate of the form (Attribute Rel_opr Value) is generated for each value in the result set. These predicates are then ANDed to form a single conjunction. An algorithmic representation of the 1-conjunction subroutine is provided in Figure 14.

```
Subroutine One_conjunction(Query_Template,Result_set,
                           Rel_opr)

   /* Query_Template:                                          */
   /*    is ((FILE = Relname) ∧  (Attrname Rel_opr Value))     */
   /* Predicate:                                               */
   /*    is (Attrname Rel_opr Value)                           */
   /* Query:                                                   */
   /*    is Query_Template ∧  Predicate ∧  . . .               */
   /*                      ∧  Predicate                        */

Strip right paren from Query_Template
Rel_opr_of_Template <-- Rel_opr

if Result_set is NOT EMPTY
  then
    Value_of_Template <-- 1st value from Result_set
    Query <-- Query_Template
      while more values in Result_set do
        Value_of_Predicate <-- next value from Result_set
        Query <-- Query || '∧ ' || Predicate
      end while
   else
     Query <-- ' '   /* Query is nil */
   Query_Template <-- Query_Template || ')'
```

Figure 14. The 1-conjunction Subroutine

Note, in Figure 14, that the template provided to the 1-conjunction subroutine is identical to that used in the N-conjunction subroutine. An additional data structure, Predicate is defined as (Attrname Rel_opr Value). The use of this additional 'template' allows us to extend the single conjunction,

((FILE = Relname) ∧ (Attrname Rel_opr Value))

to the multiple-predicate single conjunction,

((FILE = Relname) ∧ (Attrname Rel_opr Value) ∧ . . .
∧ (Attrname Rel_opr Value))

The number of predicates generated is determined by the number of values in the Result_set.

B. AN ITERATIVE STRUCTURE FOR CONTROLLING THE EXECUTION OF N-LEVEL SELECTIONS

In the previous section, we have presented algorithms for building the QUERY part of each ABDL RETRIEVE generated in the translation of a nested SQL SELECT. We now consider the process of controlling the execution of this process. An algorithmic representation of a simple structure for the control of this iterative process is provided in Figure 15. This N_level_Select subroutine is called by the Top-level process of the interface (described in Section C). The parameters passed include a series of ABDL RETRIEVE requests (in the form of a request stack), and the number, n, of such

requests. We recall, from Chapter V, that the innermost SELECT level is viewed as the nth-level. Request_Stack has the ABDL translation of the nth-level SELECT on top. The 1st-level SELECT is on the bottom. The stack is formed in this order because the nth-level request is the only request containing a fully formed query_part (as described in Chapter V). Each of the other n-1 requests requires the Result_set of the immediately preceding request before it can be sent to MDBS for processing.

```
Subroutine N_level_Select(Request_Stack,n)

  /* Request_Stack has the ABDL translation of the          */
  /* nth-level SELECT on top.  The 1st-level SELECT         */
  /* is on the bottom.  Each request in the Stack is        */
  /* composed of the reserved word RETRIEVE, Target_List,   */
  /* Set_Opr, and Query_Part.  The Query_Part of the        */
  /* nth-level SELECT is fully formed.  The Query_Part      */
  /* of the n-1 --> 1st-level SELECTs is a query template   */
  /* having the form                                        */
  /*     ((FILE = Relname) ^ (Attrname Rel_opr Value))      */
  /* with a blank in the 'Value' position.                  */

  Current_Request <-- Pop(Request_Stack)
  Send(Current_Request)
  Recieve(Result_Set)

  for i <-- 1 to n-1 do
     Current_Request <-- Pop(Request_Stack)
     Call Query_Constructor(Query_Part,Result_Set,Set_Opr)
     Send(Current_Request)
     Receive(Result_Set)
  end for
  Display(Result_Set)

end N_level_Select
```

Figure 15. An Iterative Process for Controlling the execution of N-level SELECTS

The operation of the N_level_Select subroutine is as follows. The nth-level request is popped off the top of Request_Stack and becomes the Current_Request This Current_Request is forwarded to MDBS through the Send function. Upon completion of processing, the Result_set is obtained through the Receive function. The remaining n-1 requests are popped off the stack and processed in order. The nth and succeeding result sets are incorporated into each request through a call to Query-Constructor (described in Section A). The Send and Receive functions are used on each iteration to route request/result traffic between N_level_Select and MDBS. When the last request has completed processing, the final result set is provided to the user through a call to the Display subroutine. Display presents the results of the original nested SQL SELECT as a result relation (this is the format expected by a SQL user).

C. PROPOSED SOFTWARE STRUCTURE

In this section, we present a software structure for the implementation of nested selections in our proposed SQL interface. In fact, all of the translations heretofore introduced in this thesis and in Macy [Ref. 8], are supported by this structure. Therefore, allowing for possible modifications required to support additional multiple and single-relation SQL operations, the software structure depicted in Figure 16 represents the overall software structure of the SQL interface.

As depicted in Figure 16, the SQL interface is comprised of a single top-level process with multiple subroutines and functions. The top-level process is called SQLI (SQL Interface). We have described the N_level_Select subroutine

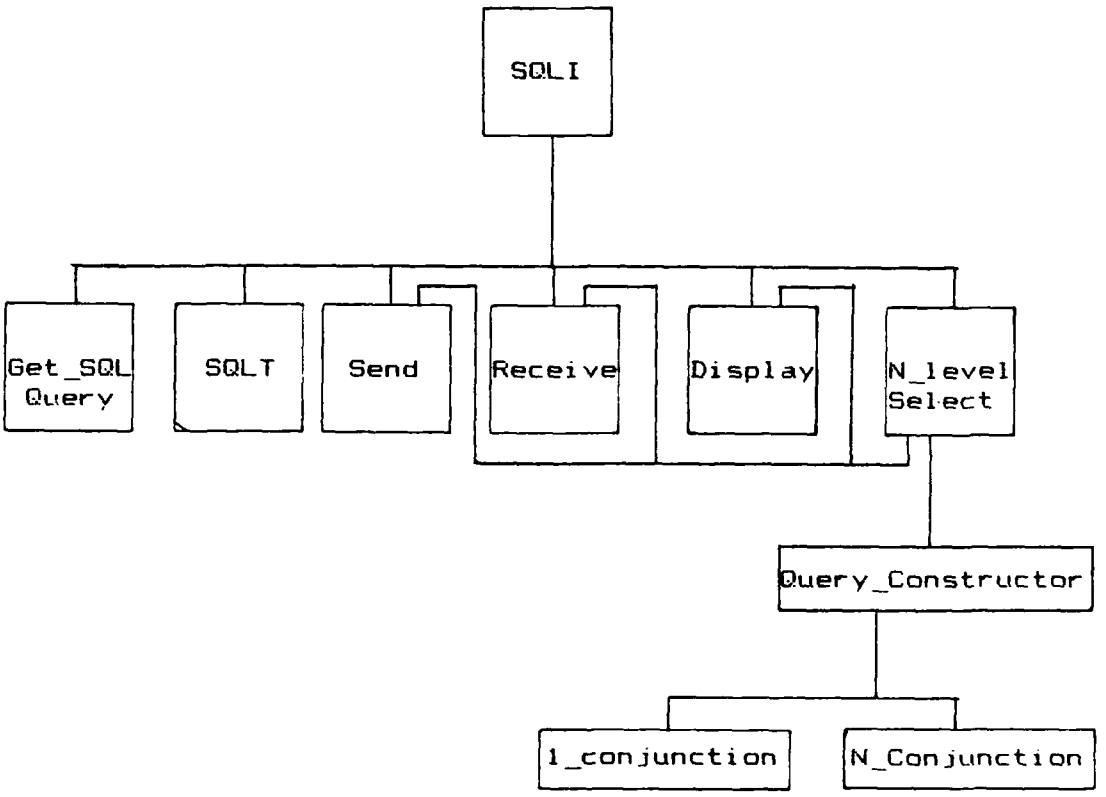


Figure 16. The Proposed Software Structure

group. We discuss the remaining subroutines as we explain the functionality of SQLI. An algorithm for SQLI is presented in Figure 17.

The operation of SQLI is an follows. Once a session is initiated from the user terminal, the actions depicted in ALGORITHM SQLI are repeated until session termination. The SQL query to be translated into the equivalent ABDL construct is obtained through a call to the subroutine Get_SQL_Query. This subroutine polls the user terminal for

```
ALGORITHM SQLI

 Repeat
   CALL Get_SQL_Query(Query)
   CALL SQLT(Query,Request_Stack,N,Errors)
   if N = 0 then    /* Syntax Errors */
     CALL Display(Query)
     CALL Display(Errors)
   else if N = 1 then    /* Single Request */
     Send(Pop(Request_Stack))
     Receive(Result_Set)
     CALL Display(Result_Set)
   else                          /* N-level Request */
     CALL N_level_Select(Request_Stack,N)
   end if
  End_of_session?
 until end_of_session
end ALGORITHM SQLI
```

Figure 17. The Top-level Process of the Interface, SQLI

input. Note that when a query is obtained, the polling stops until the result relation is received by the user (or syntax errors are displayed for the user). This restriction is placed in order to preclude the complexity of processing more than one request at a time. (We assume that several user terminals have access to a copy of SQLI, and that each user makes a request and waits for a result before making another request).

The query obtained by the call to Get_SQL_Query is passed as a parameter in a call to the SQL Translator (SQLT) subroutine. SQLT parses the query, recognizes the query-type, checks for syntax errors, and translates the SQL query to the appropriate ABDL request. If there are no syntax errors, SQLT places the translated requests in a stack and returns this Request_Stack, along with the number, N, of requests in the stack. In the case of simple, single-relation operations, Request_Stack contains one request. In the case of a nested selection, SQLT first parses and translates the outermost SELECT placing the resultant RETRIEVE request on the stack. As previously discussed, the request contains a query-template. (Recall that only the nth-level, or innermost, request is fully formed). If there are syntax errors, SQLT returns a value of zero for N. The errors are also returned.

If the number of requests in Request_Stack is zero (N = 0), then SQLT has detected syntax errors. In this case, SQLI makes two calls to the Display subroutine in order to provide the user a display of the query and of the errors detected. If the number of requests in Request_Stack is one (N = 1), then the single request is popped off the stack and forwarded, via the Send function, to MDBS for processing. The Result_set is obtained through the Receive function. The result relation is provided to the user through a call to the Display subroutine. If the number of requests in

Request_Stack is greater than one, then N_level_Select is called. The subsequent processing is explained in Section B.

As previously discussed, we propose that the SQL interface be implemented such that SQLI and its subroutines are resident on a host computer. This precludes the need to place an additional workload on the MDBS Controller. In effect, MDBS is "unaware" that the user is making database requests in SQL, and the user need only know what information is desired and how to form the request in the syntax of SQL. The logical structure of the system is depicted in Figure 18.

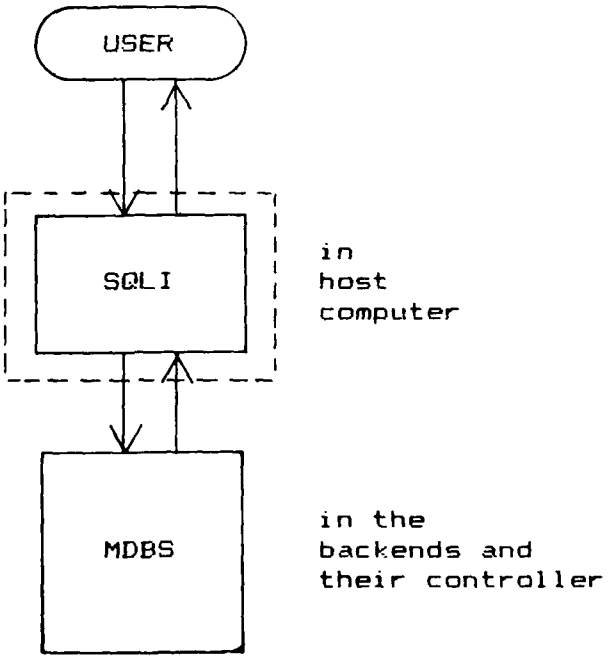


Figure 18. The Logical Structure of the System

VII. ADDITIONAL SQL-TO-ABDL TRANSLATIONS

We have described single-relation set membership and multiple-relation nested SQL SELECT operations. For each SQL operation, we have developed the appropriate ABDL translation. In Chapter VI, we have proposed a software structure to facilitate the implementation of these translations, in addition to the simple, single-relation translations which Macy [Ref. 8] has provided. In this chapter, we investigate other selected single-relation and multiple-relation SQL operations. Inclusion of these highly desirable options in the SQL set operations supported by the interface further demonstrates the power of ABDL to support relational operations.

As in previous chapters, the approach of this chapter is to describe each SQL operation and then determine which ABDL constructs can be used to support the operation. As each translation is developed, we show graphically, algorithmically, and through text how the software structure of the interface (described in Chapter VI) must evolve in order to accomodate the additional operations. The single-relation operations are presented in Section A, and the multiple-relation operations are presented in Section B. In Section C, we present the modified software structure of the SQL interface.

A. SELECTED SINGLE-RELATION OPERATIONS

The single-relation operations selected for discussion in this section include: updating multiple attributes in a single record; retrieving groups of attributes which satisfy a group condition; retrieving computed values; providing format options; retrieving ordered attributes (SORT); and eliminating duplicates (PROJECTION). These operations are commonly supported in commercial relational database systems utilizing the SQL language. A SQL-trained user of the interface proposed in this thesis would expect to be able to utilize familiar SQL constructs to perform these operations. We address the SQL-TO-ABDL translations in the following subsections.

1. Updating Multiple-Attributes

All data languages provide a data update capability. Of interest here is the SQL construct for update. This construct allows the user to change the values of any number of attributes stored in the record by issuing a single query. This capability is both convenient and efficient. The following example depicts the updating of multiple-attributes (fields) in a single record. If we wish to change the color of part P2 to yellow, increase its weight by 5, and set its city to Normandy, we may use the following SQL query:

```
UPDATE   P
SET      COLOR  = 'Yellow',
         WEIGHT = WEIGHT + 5,
         CITY   = 'Normandy'
WHERE    P#     = 'P2'
```

In this example, we are updating the attributes COLOR, WEIGHT, and CITY in a single record with primary key, P2. The record is contained in the Parts (P) relation. Note that any reference to an attribute on the right-hand side of an equals sign refers to the value of that attribute prior to updating.

In studying the SQL example above, we note that there are three cases to consider depending on the attributes listed in the SET and WHERE clauses. We refer to these as case-0, case-1, and case-2 updates. To facilitate the following explanation, let S be the set of distinct attribute names listed in the SET clause, and W be the set of distinct attribute names listed in the WHERE clause. In case-0 updates (e.g., the above example), no attribute is listed in both the SET and WHERE clauses (i.e., $S \cap W = 0$). In case-1 updates, one attribute is listed in both clauses (i.e., cardinality($S \cap W$) = 1). In case-2 updates, multiple attributes are listed in both clauses (i.e., cardinality($S \cap W$) > 1). A case-1 modification of our example is as follows:

```
UPDATE    P
SET       COLOR  = 'Yellow',
          WEIGHT = WEIGHT + 5,
          CITY   = 'Normandy'
WHERE     (P# = 'P2') AND (CITY = 'Paris')
```

Note, CITY is in both S and W, and the cardinality of (S ∩ W) is 1. A case-2 modification of our original example is as follows:

```
UPDATE    P
SET       COLOR  = 'Yellow',
          WEIGHT = WEIGHT + 5,
          CITY   = 'Normandy'
WHERE     (P# = 'P2') AND (CITY = 'Paris')
                      AND (COLOR = 'GREEN')
```

Note, CITY and COLOR are in both S and W, and cardinality(S ∩ W) > 1. The SQL-to-ABDL translations of the three cases of multiple-attribute update are described in the following subsection.

a. The translation to ABDL

ABDL does not provide a single-request construct which updates more than one attribute in a record. We must translate the SQL UPDATE into multiple ABDL UPDATEs. Case-0 SQL UPDATE queries can be translated directly to multiple ABDL UPDATE requests. The order in which these requests are processed is immaterial. The case-0 example above

translates to the following three independent ABDL UPDATE requests:

```
UPDATE ((FILE = P) ∧ (P# = P2)) (COLOR = Yellow)
UPDATE ((FILE = P) ∧ (P# = P2)) (WEIGHT = WEIGHT + 5)
UPDATE ((FILE = P) ∧ (P# = P2)) (CITY = Normandy)
```

Our case-1 example translates to the same three UPDATE requests, however, the presence of the CITY attribute in both the WHERE and SET clauses effects the structure of the translation. The order of request processing now becomes important. For example, if CITY is updated first, the condition ((P# = 'P2') AND (CITY = 'Paris')) is no longer satisfied when a subsequent attempt is made to process the COLOR and WEIGHT UPDATE requests. ABDL provides a construct called a Transaction which specifies the order in which a series of requests must be processed. Therefore, the case-1 translation becomes

```
BEGIN Transaction
   UPDATE ((FILE = P) ∧ (P# = P2)) (COLOR = Yellow)
   UPDATE ((FILE = P) ∧ (P# = P2)) (WEIGHT = WEIGHT + 5)
   UPDATE ((FILE = P) ∧ (P# = P2)) (CITY = Normandy)
END Transaction
```

Requests within a transaction are processed in the same order as they are specified. Therefore, we can obtain a correct result.

The case-2 example also translates to a series of three ABDL requests. However, the translation is more complex. In this case, multiple attributes specified in the WHERE clause are also listed in the SET clause. When the first of these attributes is updated, all subsequent attempts to update the remaining attributes will fail. Since the WHERE condition is no longer satisfiable, the record can not be found. The following sequence of ABDL requests accomplishes the requested update. (Note that the ABDL UPDATE construct is not used).

```
RETRIEVE ((FILE = P) ∧ (P# = P2)) <P#,PNAME,COLOR,WEIGHT,
                                   CITY>

DELETE ((FILE = P) ∧ (P# = P2) ∧ (PNAME = Bolt) ∧
        (COLOR = Green) ∧ (WEIGHT = 17) ∧
        (CITY = Paris))

INSERT (<FILE = P>,<P# = P2>,<PNAME = Bolt>,
        <COLOR = Yellow>,<WEIGHT = 22>,<CITY = Normandy>)
```

b. A proposed Software Structure

In order to implement multiple-attribute updates, we must augment the functionality of the software structure (SQLI) which we have developed in Chapter VI. We specify an additional parameter, Request_Type, to be returned by SQLT. When the value of Request_Type is 'Case0_update', the subroutine Case0_update is called. In this case, the multiple ABDL RETRIEVE requests are simply removed from Request_Stack and forwarded to MDBS for processing. As previously stated, the order of processing

does not effect the result. When all updates are complete, the user is so informed. When the value of Request_Type is 'Case1_update', the subroutine, Case1_update is called. When the value of Request_Type is 'Case2_update',the subroutine, Case2_update is called. The Case1_update and Case2_update subroutines are presented in Figures 19 and 20, respectively.

```
Subroutine Case1_update(Request_Stack,Result_Set)

  /* Transaction_Request:                                    */
  /*    is a template with the Reserved word BOT             */
  /*    followed by multiple blank lines (to be used         */
  /*    by the series of requests) and the Reserved          */
  /*    word EOT.                                            */

 while NOT EMPTY(Request_Stack) do
   Pop(Request_Stack)
   Fill in blank lines of Transaction_Request with
      requests from Request_Stack
 end while
 Send(Transaction_Request)
 Receive(Result_Set)          /* Result_Set returned to     */
                              /* calling routine            */

end Case1_update
```

Figure 19. Subroutine Case1_Update

The Case1_Update subroutine builds a transaction of update requests for MDBS processing. The subroutine is provided the parameter Request_Stack which contains multiple UPDATE requests stacked such that the request on the bottom of the stack is the request which must be processed last.

Subroutine Case1_Update sends the request transaction to MDBS, Receives the Result_Set, and returns the Result_Set to the calling routine.

```
Subroutine Case2_update(Request_Stack,Result_Set)

  /* Insert_Template:                                          */
  /*   is the INSERT request with values for the               */
  /*   attributes-to-be-updated and blanks for the             */
  /*   attributes whose values are obtained by the             */
  /*   RETRIEVE request.                                       */

Send(Pop(Request_Stack))          /* RETRIEVE request          */
Receive(Result_Set)
Send(Pop(Request_Stack))          /* DELETE the record         */
Receive(Result_Set)               /* deletion is complete      */
While  there are records to update  do
  Insert_Template <-- /* fill in blanks with retrieved       */
                      /* attribute values                    */
  Insert_Request <--  /* form the INSERT request from        */
                      /* the record and Insert_Template      */
  Send(Insert_Request)
end while
Receive(Result_Set)               /* INSERT is complete        */

end Case2_update
```

Figure 20. Subroutine Case2_update

The Case2_Update subroutine controls the execution of the RETRIEVE-DELETE-INSERT series of requests. The RETRIEVE obtains a copy of the appropriate record(s). The DELETE deletes the original record(s) in the database. The INSERT re-inserts the record(s) with all the modified attribute values.

2. Retrieving Qualified Groups

Both SQL and ABDL provide an option whereby retrieved attributes may be grouped. For example, if we

wish to obtain the part number and the total quantity for each part supplied, we may utilize the following SELECT construct:

```
SELECT  P#,SUM(QTY)
FROM    SP
GROUP   BY P#
```

The result relation is:

P#	
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

Note that "...each expression in the SELECT clause must be single-valued for each group; that is, it can be either the GROUP_BY field itself, or a function such as SUM that operates on all values of a given field within a group and reduces those values to a single value." [Ref. 9]

The above SQL operation is directly supported by the software structure of Chapter VI. Using the SELECT-to-RETRIEVE mapping which we have described in Chapter III, the equivalent ABDL construct is:

```
RETRIEVE (FILE = SP)<P#,SUM(QTY)> BY P#
```

SQL provides a further option for use with grouped attributes. Once the rows of a table are grouped by a selected attribute, groups not satisfying a specified

condition can be eliminated through the use of the HAVING operator. The following comprehensive example clarifies the use of the 'GROUP BY with HAVING' option. If we wish to obtain the part number and the maximum quantity of the part supplied for all parts such that the total quantity supplied is greater than 300 (excluding from the total all shipments for which the quantity is less than or equal to 200), we may use the following query:

```
SELECT   P#,MAX(QTY)
FROM     SP
WHERE    QTY > 200
GROUP    BY P#
HAVING   SUM(QTY) > 300
```

We can imagine the result relation

P#	
P1	300
P2	400
P3	400
P5	400

being formed as follows. A copy is made of table SP (FROM). The rows not satisfying "QTY > 200" are eliminated (WHERE). The remaining rows are then grouped by P# (GROUP BY). The newly formed groups are checked against the predicate "SUM(QTY) > 300". Those not satisfying the condition are eliminated (HAVING). Finally, part numbers and maximum quantities are extracted from the remaining groups (SELECT).

a. The Translation to ABDL

As previously discussed, ABDL provides a construct for the retrieval of data which is grouped by a selected attribute. In the comprehensive SQL example above, the use of the HAVING operator specifies a further qualification on the groups. In this example, the groups whose total quantity supplied is less than or equal to 300 are to be eliminated. ABDL does not provide a facility for checking this group condition. This condition must be checked in the interface. The SQL query is translated to the ABDL request

```
RETRIEVE ((FILE = SP) ∧ (QTY > 200)) <P#,MAX(QTY),SUM(QTY)>
                                                    BY P#
```

which we imagine returns the following table:

P#	MAX(QTY)	SUM(QTY)
P1	300	600
P2	400	400
P3	400	400
P4	300	300
P5	400	400

Software in the interface then checks the HAVING condition "SUM(QTY) > 300". This eliminates the grouping for part P4. The remaining part numbers and maximum quantities are returned to the user.

b. A Proposed Software Structure

When SQLT returns the value, 'Group_by_having' for the parameter, Request_Type, we assume that the HAVING condition is also made available to the Group-By-Having subroutine. (We make a similar assumption for other Request_Types). The subroutine sends the request, receives the result set, checks the HAVING condition, and returns only those tuples satisfying the having condition to the user. Figure 21 depicts this operation.

```
Subroutine Group-By-Having(Request_Stack,HAVING_condition,
                           Result_Set)
  Send(Pop(Request_Stack)
  Receive(Result_Set)
  Eliminate groups not satisfying HAVING condition

end Group_By_Having
```

Figure 21. Subroutine Group_By_Having

3. Retrieving Computed Values

The concept of retrieving computed values is simple, yet it typifies the important options that database management system designers are providing in order to ensure user-friendliness and user-flexibility. This option supports the inclusion of arithmetic expressions involving fields as well as simple field-names. For example, the user should be able to specify units-of-measure for numerical results. SQL supports this concept. If we wish to obtain the part number and the weight of the part in grams (given in table P in

pounds), we may use the following query:

```
SELECT P#,WEIGHT * 454
FROM   P
```

The result relation is:

P#	
P1	5448
P2	7718
P3	7718
P4	6356
P5	5448
P6	8626

a. The Translation to ABDL

In this translation, the ABDL request retrieves the indicated attributes leaving any computation to be accomplished in the interface. For the example above, the ABDL translation is

```
RETRIEVE (FILE= P) <P#,WEIGHT>
```

The specified arithmetic operation is performed by interface software on the retrieved values for WEIGHT (i.e., WEIGHT * 454) prior to returning the final result relation to the user. The software required is a simple interpreter for evaluating arithmetic expressions.

b. A Proposed Software Structure

An Expression_Evaluator subroutine can be used to accomplish the arithmetic operations specified in the SQL query. The subroutine simply utilizes the appropriate function (e.g., Mult,Add,Sub,Div) to perform the operation.

4. Providing Format Options

Often, the information retrieved from a database is intended for use in published reports. The availability of formatting options can make generating these reports simpler. For example, while it is prudent to save disk space by storing the names of suppliers as values for an attribute-name such as SNAME, an end-user unfamiliar with the database is psychologically more comfortable with a column heading such as SUPPLIERS. In SQL queries, the desired format is indicated in the SELECT clause. For example, if we wish to obtain the names of all suppliers, we may use the following query:

```
SELECT  SNAME SUPPLIERS
FROM    S
```

The result relation is:

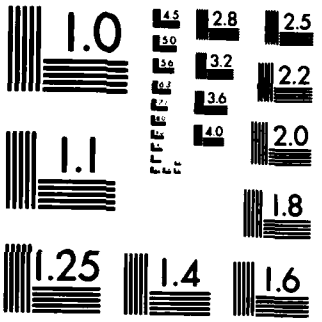
SUPPLIERS
Smith
Jones
Blake
Clark
Adams

Note that the column heading is SUPPLIERS rather than the field name, SNAME.

a. The Translation to ABDL

This translation is similar to that presented in Subsection 2 above. Information, returned from MDBS, is modified by the interface software. The SQL SELECT query is translated to the ABDL request

AD-A151 501 DESIGN AND ANALYSIS OF A COMPLETE RELATIONAL INTERFACE 2/2
FOR A MULTI-BACKEND DATABASE SYSTEM(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA R E ROLLINS JUN 84
UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RETRIEVE (FILE = S) <SNAME>

The results of this request are modified by the SQL interface (SQLI) prior to returning the final result relation to the user. In this case, the column heading, SNAME, is changed to the new heading, SUPPLIERS.

b. A Proposed Software Structure

Format options can be provided in the Display subroutine. Any change in the form of the table heading can be passed at the time of the call to Display.

5. The Retrieval with Ordering (SORT)

Generally, the result of a SELECT operation is not guaranteed to be in any particular order. Ordering (SORT) is normally not accomplished in SQL queries unless specifically requested by the user. This operation may be costly, and the additional expense is often unwarranted. In SQL, the user may specify ordering through the use of the ORDER_BY operator. As an example, if we wish to obtain supplier-numbers for all suppliers providing shipments, such that the result is ordered by supplier-number, we may use the following query:

```
SELECT  UNIQUE S#
FROM    SP
ORDER   BY S#
```

The result relation is:

S#
S1
S1
S1
S1
S1
S1
S2
S2
S3
S4
S4
S4

a. The Translation to ABDL

In the translation of the above SQL query, we assume an ordering capability within MDBS. The development of this capability is the goal of a current thesis by Muldur [Ref 15]. The ABDL request

RETRIEVE (FILE = SP) <S#> ORDER BY S#

returns all supplier numbers (ordered by increasing supplier numbers) contained in the SP file (including duplicates).

b. A Proposed Software Structure

We assume that the ordering of selected attributes is directly supported by MDBS. Therefore, no augmentation of SQLI is required.

6. An Elimination of Duplicates (PROJECTION)

The results of a SELECT operation may contain duplicates. The elimination of duplicates (PROJECTION), as in the case of retrieval with ordering (SORT), is normally not accomplished in SQL queries unless specifically

requested. Again, the cost is high and often unwarranted. An exception to this rule is that duplicate rows are automatically eliminated in UNION operations. (UNION operations are described in Section B).

In SQL, the elimination of duplicates may be specified through the use of the UNIQUE operator. As an example, if we wish to obtain supplier-numbers for all suppliers providing shipments such that no supplier-number is listed more than once, and the result is ordered by supplier-number, we may use the following query:

```
SELECT  UNIQUE S#
FROM    SP
ORDER   BY S#
```

The result relation is:

S#
S1
S2
S3
S4

This example is a modification of the example presented in Subsection 5. Note that duplicate supplier-numbers are eliminated.

a. The Translation to ABDL

The ABDL translation for the above SQL query is identical to the translation for our Subsection 5 example. Again, the ABDL request

```
RETRIEVE (FILE = SP) <S#> ORDER BY S#
```

returns all supplier-numbers (ordered by increasing supplier-numbers) contained in the SP file (including duplicates). Since UNIQUE is specified in the SELECT clause of the SQL query, SQLI must check the ordered rows eliminating duplicate values for the S# attribute prior to forwarding the result relation to the user. If our example is modified such that the ORDER BY clause is omitted, we may facilitate the elimination of duplicates by "forcing" a SORT of the selected attributes. That is, the ABDL RETRIEVE request is written to include an ORDER BY specification.

b. A Proposed Software Structure

When UNIQUE is specified in the SQL query, the Result_Set from MDBS is passed in a call to a Duplicate_Eliminator subroutine. This subroutine scans and compares adjacent members of an ordered Result_Set eliminating duplicate members. We assume that the Result_Set is always ordered prior to being passed to Duplicate_Eliminator. The ordering is either user-specified or "forced" in the SQLT translation.

B. SELECTED MULTIPLE-RELATION OPERATIONS

In this section, we discuss two additional multiple-relation operations which are supported by SQL: retrieval using the UNION operator and retrieval specifying JOIN operations. These two operations and the nested SELECT (described in Chapter V) give SQL much of its power and flexibility. The availability of query constructs which

allow access to related data in multiple tables greatly enhances the ease with which a user can obtain the desired information from the database. We investigate UNION and JOIN operations in the following subsections.

1. The Retrieval Using UNION

From set theory, we recall that the UNION of sets A and B (i.e., A UNION B) is the set of all objects x such that x is a member of A or x is a member of B. The formal predicate logic definition of A UNION B is:

$$\forall x \, [(x \in A) \vee (x \in B)]$$

In SQL, the UNION operator is used in a query comprised of multiple-SELECT constructs. As an example, if we wish to obtain numbers for parts that either weigh more than 16 pounds or are currently supplied by supplier S2 (or both), we may use the following query:

```
SELECT  P#
FROM    P
WHERE   WEIGHT > 16

UNION

SELECT  P#
FROM    SP
WHERE   S# = 'S2'
```

The result relation is:

P#
P2
P3
P6
P1

From the sample database of Chapter I, we can see that parts P2, P3, and P6 weigh more than 16 pounds ($x \in A$). Part P1 weighs less than 16 pounds, however, P1 is currently supplied by supplier S2 ($x \in B$). Part P2 weighs more than 16 pounds and is supplied by supplier S2 ($(x \in A) \wedge (x \in B)$). Note that duplicate rows are eliminated from the result of a UNION operation.

a. The Translation to ABDL

In the SQL query above, each SELECT construct translates into an equivalent ABDL RETRIEVE request. In this example, the two ABDL requests

```
RETRIEVE (FILE = P) ∧ (WEIGHT > 16) <P#> ORDER BY P#
RETRIEVE (FILE = SP) ∧ (S# = S2) <P#> ORDER BY P#
```

are processed concurrently. The results are combined in SQLI, where duplicate rows are eliminated. The remaining rows are forwarded to the user.

b. A Proposed Software Structure

When the value of Request_Type is UNION, the translation and processing are as follows. An MDBS SORT is specified in the ABDL translation. A subroutine called UNION pops all ABDL RETRIEVE requests off of Request_Stack

and forwards them to MDBS for concurrent processing. The ordered result sets are merged (through the use of a standard merge function), and then passed to Duplicate_Eliminator. Finally, the uniquely selected results of the UNION operation are returned to SQLI for display to the user. Subroutine UNION is presented in Figure 22.

```
Subroutine UNION(Request_Stack,Result_Set)

 while NOT EMPTY(Request_Stack) do
   Send(Pop(Request_Stack)
 end while
 Receive(Result_Set1)
 Receive(Result_Set2)
 Merge(Result_Set1,Result_Set2)
 CALL Duplicate_Eliminator(Result_Set)

end UNION
```

Figure 22. Subroutine UNION

2. The Retrieval Specifying Join Operations

Join operations are characteristic of data languages intended for use with relational databases. SQL provides the capability to specify implicit join, equality join, and inequality join operations. In an implicit join, attribute-values in multiple tables are compared, however, the values returned to the user are taken from only one table. Implicit joins can be formed through the use of the nested SQL SELECT constructs which we have described in Chapter V. In the nested SELECT, multiple tables are accessed and the

values of selected attributes are compared. We note that only values from the outermost SELECT are returned in the final result set. This operation results in the formation of an implicit join.

Equality join and inequality join operations are specified by referencing multiple tables in a single SELECT query. As an example of an equality join, if for each part supplied we wish to obtain part numbers and names of all cities supplying the part, we may use the following query:

```
SELECT  UNIQUE  P#,CITY
FROM    SP,S
WHERE   SP.S# = S.S#
```

The result relation is:

P#	CITY
P1	London
P1	Paris
P2	London
P2	Paris
P3	London
P4	London
P5	London
P6	London

Note that table-names may be used as qualifiers in the SELECT and WHERE clauses in order to resolve ambiguities or to ensure clarity. For example, the SELECT clause may be equivalently written

```
SELECT  UNIQUE  SP.P#,S.CITY
```

Although there are optimization techniques which facilitate a more efficient implementation, we can visualize the join operation as follows. First the Cartesian product of SP and S is formed. Then, rows not satisfying the condition SP.S# = S.S# are eliminated. Next, columns P# and CITY are projected from the remaining rows. Finally, since the operator UNIQUE is used, all duplicate rows are removed before the result relation is returned to the user. (For an indepth discussion of the efficiency and optimization considerations of implementing join operations, the reader is referred to Demurjian [Ref. 1]).

a. The Translation to ABDL

The attribute-based data language, as implemented in MDBS, does not provide a join capability. Muldur [Ref. 15] is currently investigating the practicality of incorporating join operations within MDBS. If we assume that the functionality of MDBS is augmented to support the equality join and inequality join operations, we might use the following translation for the equality join (as discussed in Demurjian [Ref. 1]). The general form of a simple, two-way equality join expressed in the syntax of SQL is

```
SELECT  sel_expr_list
FROM    relation_name1, relation_name2
WHERE   relation_name1.attribute = relation_name2.attribute
        AND qualification
```

The general form of the ABDL translation is

```
RETRIEVE     (attribute_list_1) (query_1)
CONNECT ON   (attribute_1, attribute_2)
             (attribute_list_2) (query_2)
```

The sel_expr_list of the SQL SELECT is divided into a target list consisting of attributes from relation_name1 and a target list consisting of attributes from relation_name2. The qualification of the SQL SELECT is likewise partitioned. The attributes named in the equality predicate become the object of the CONNECT ON clause in the ABDL request. Following this general form, the translation for the equality join example of the preceding subsection is

```
RETRIEVE  < (S#,P#) (FILE = SP) >
CONNECT ON (SP.S#, S.S#)
          < (S#,CITY) (FILE = S) >
```

b. A Proposed Software Structure

As stated previously, we assume a join capability for MDBS. Therefore, no augmentation of SQLI is required.

C. THE MODIFIED SOFTWARE STRUCTURE OF THE SQL INTERFACE

In this section, we present the modified software structure of SQLI. We modify the structure which we have presented in Chapter VI in order to facilitate the implementation of the additional operations described in

this chapter. The modified version of the top-level process, SQLI, is shown in Figure 23. Note, we have simplified this algorithm through the use of the Request_Control subroutine. The functionality of this subroutine is presented in Figure 24. The purpose of Request_Control is to provide overall control of request processing for the interface. A high-level view of the modified software structure is shown in Figure 25, and the relationship between Subroutine Request_Control and its subordinate group of subroutines is depicted in Figure 26.

```
ALGORITHM SQLI (Modified)

 Repeat
   CALL Get_SQL_Query(Query)
   CALL SQLT(Query,Request_Stack,N,Errors,Request_Type,
             Format_Option,Arith_Expr)
   if N = 0 then                /* Syntax Errors */
     CALL Display(Query)
     CALL Display(Errors)
   else
   CALL Request_Control(Request_Stack,N,Request_Type,
                        Arith_Expr,Result_Set)
   CALL Display(Result_Set,Format_Option)
   end if
 End_of_session?
 until end_of_session
end ALGORITHM SQLI (Modified)
```

Figure 23. ALGORITHM SQLI (Modified)

```
Subroutine Request_Control(Request_Stack,N,Request_Type,
                           Arith_Expr,Result_Set)
 CASE Request_Type OF

 Case0_Update:   CALL Case0_Update(Request_Stack,Result_Set);

 Case1_Update:   CALL Case1_Update(Request_Stack,Result_Set);

 Case2_Update:   CALL Case2_Update(Request_Stack,Result_Set);

 Group_Having:   CALL Group_Having(Request_Stack,
                                   Condition,Result_Set);

 UNION:          CALL UNION(Request_Stack,Result_Set);

 Others:         if N = 1 then

                   CALL One_Request(Request_Stack,Result_Set)
                        /* for simple, directly-supported  */
                        /* single request                  */
                 else
                   CALL N_Level_Select(Request_Stack,
                                       N,Result_Set)
                 end if

 END CASE
```

Figure 24. Subroutine Request_Control

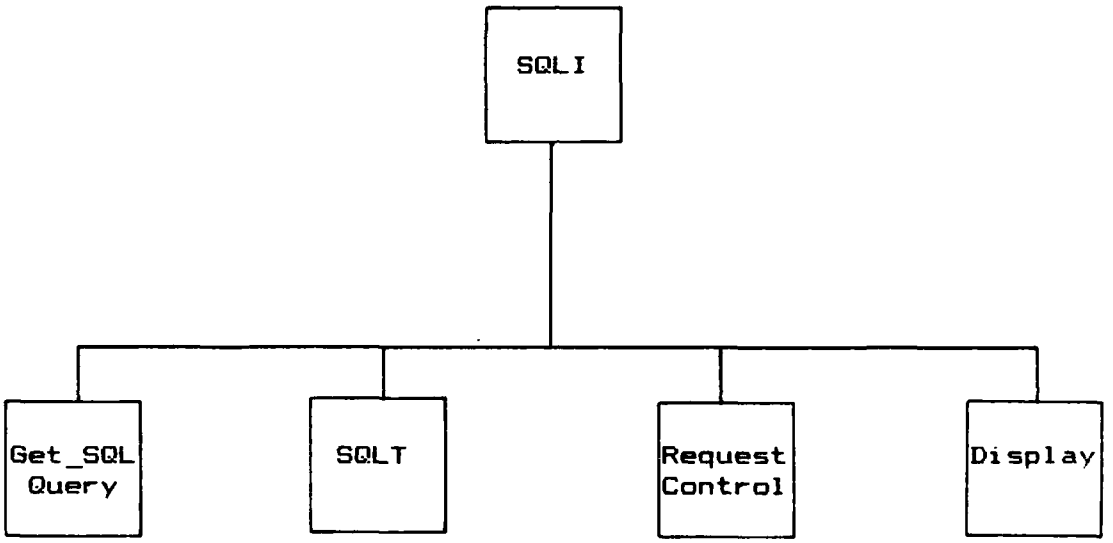


Figure 25. A High-Level View of the Software Structure

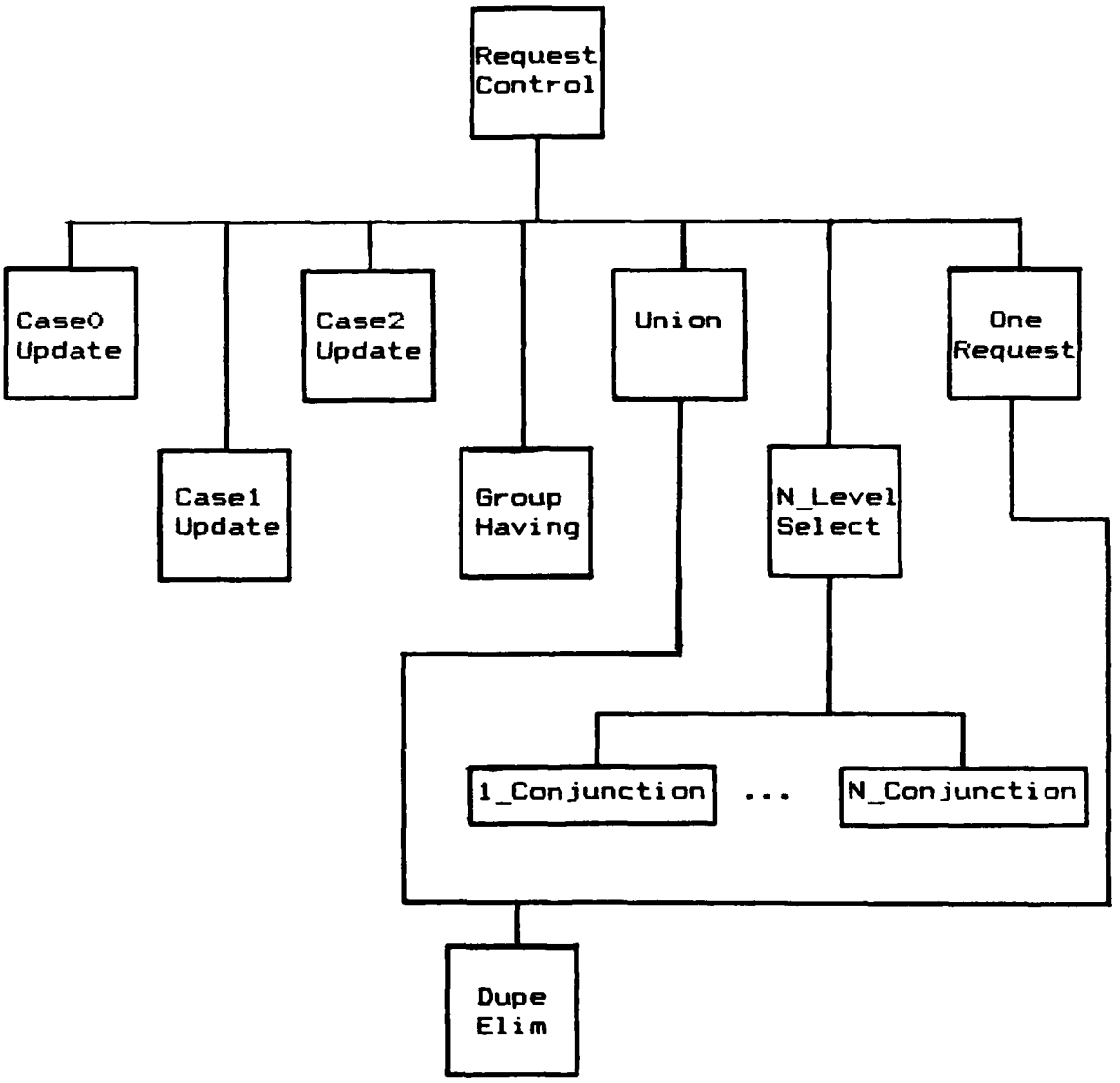


Figure 26. Request_Control and its Subroutines

VIII. CONCLUDING REMARKS

In this thesis, we have concentrated on the language interface aspects of using an attribute-based database system, MDBS, as a kernel for the support of the relational data model and the relational query language, SQL. A related thesis by Weishar [Ref. 16] provides the design and analysis of an interface for the hierarchical model and the hierarchical data language, DL/I. This work is part of ongoing research being conducted by the Laboratory for Database Systems Research under the direction of Dr. David K. Hsiao. As stated in [Ref. 1], the goal of this phase of the laboratory's research "...is to provide increased utility in database computers. A centralized repository of data is made available to multiple, dissimilar hosts. Furthermore, the database is also made available to transactions written in multiple, dissimilar data languages."

The rapid evolution of database technology has provided the motivation for this research. Commercial database management systems have only been available since the 1960's. Today, organizations of all types are critically dependent on the operation of these systems. This dependency comes from the need to centrally control large

quantities of operational data. The information must be accurate and readily accessible by relatively inexperienced end-users.

There are three generally known approaches to the design of database systems. These are the network, hierarchical, relational approaches. An organization normally chooses a commercial system based on one of these models. The database must be created and operator and user personnel must be trained. Because of the re-programming and re-training effort (and money) required, an organization is unlikely to change to a system based on one of the other models.

We have discussed an alternative to the development of separate stand-alone systems for specific data models. In this proposal, the three generally known models and their model-based data languages are supported by the attribute-based data model and data language. We have shown (in the relational case) how a software interface can be built for such support.

Specific contributions of this thesis include extremely thorough explanations of SQL operations such as: set-membership, nested retrievals, retrieval of grouped attributes, join operations, retrieval of computed values, providing format options, retrieval using UNION, updating multiple fields, retrieval with ordering, and elimination of duplicates. We have extended the work of Macy [Ref. 8] by

showing that many of the SQL constructs for the above operations are directly supportable by ABDL and MDBS. Others can be translated into a series of the primary and aggregate operations of the attribute-based system. In all cases, SQL-to-ABDL translations are provided. We have also proposed a software structure to facilitate the future implementation of the SQL interface.

A major design goal has been to design a SQL interface to MDBS without requiring that changes be made to the MDBS system. We have shown that the complete interface can be implemented on a host computer. All translations are accomplished in the SQL interface. MDBS continues to receive and process requests written in the syntax of ABDL. We have also shown that the interface can be designed to utilize existing ABDL constructs (either one or a series of ABDL requests). No changes to the ABDL syntax are required. We also have not proposed any changes to the syntax of SQL. We have designed the interface to be transparent to the SQL user. The intention is that a trained SQL user need know nothing of the existence of the interface or of MDBS. The user can log in at a system terminal, input a SQL query, and obtain result data in a relational format.

In retrospect, our unconventional bottom-up approach to design seems entirely appropriate. We have built upon the basic subset of SQL-to-ABDL mappings provided by Macy [Ref. 8], making additions to the set as selected SQL operations

have been incorporated into the interface. As our investigation begins in Chapter IV, the form of the interface software structure is not clear. When the nested SQL SELECT is described in Chapter V, the requirements for the structure begin to solidify. We become aware that an iterative structure is needed to control the processing of series of ABDL requests. As the algorithm, SQLI, is completed in Chapter VI, it is clear that we have developed the overall software structure for the SQL interface. The functionality of the structure is enhanced as additional SQL operations are selected. However, the general structure remains intact.

As an alternative to implementing the SQL (network and hierarchical, as well) interface on a host computer, the interface can be placed inside of MDBS. We have studied this possibility, and recommend against such an implementation. A major design goal of MDBS is to minimize the role of the controller. This goal can not be attained if the controller must support the operation of resident relational, network, and hierarchical interfaces.

We have shown that the attribute-based system supports relational database applications. We have provided SQL-to-ABDL translations for selected database operations, and we have proposed a software structure to facilitate implementation. The next step is to implement the interface on a host computer. In order to finally determine the

overall practicality of using MDBS as a kernel database system, we must also implement the hierarchical interface design of Weishar [Ref. 16]. Additionally, an interface to support the network model must be designed and implemented.

APPENDIX A: FORMAL SPECIFICATION OF THE ATTRIBUTE-BASED DATA LANGUAGE, ABDL

The following is the BNF for the attribute-based data language developed by Hsiao and Menon [Refs. 4 and 10]. Square brackets [ ] are used to indicate optional constructs.

```
Predicate                 :=  attribute rel_op value

attribute                 :=  char_string

attribute_being_modified  :=  attribute

base_attribute            :=  attribute

value                     :=  string
                              | number
                              | float

Conjunct                  :=  (Predicate)
                              | (Conjunct /  Predicate)

Query                     :=  Conjunct
                              | Query /  Conjunct

Stat                      :=  AVG | MAX | MIN | SUM | COUNT

list_el                   :=  Stat (attribute)

list                      :=  attribute
                              | list_el
                              | list,attribute
                              | list,list_el

Target_list               :=  (list)

Attrib_val_pair           :=  <attribute,value>

Half_record               :=  Attrib_val_pair
                              | Half_record, Attrib_val_pair

Record                    :=  (Half_record)
```

```
Pointer                 :=  number
Modifier                :=  type-0
                            | type-I
                            | type-II
                            | type-III
                            | type-IV

type-0                  :=  <attribute_being_modified =
                             value>

type-I                  :=  <attribute-being_modified =
                             expr1>

type-II                 :=  <attribute_being_modified =
                             expr2>

type-III                :=  <attribute_being_modified =
                             expr2 of Query>

type-IV                 :=  <attribute_being_modified =
                             expr2 of Pointer>

Request                 :=  Insert | Delete | Update |
                            Retrieve

Insert                  :=  INSERT Record

Delete                  :=  DELETE Query

Update                  :=  UPDATE Query Modifier

Retrieve                :=  RETRIEVE Query Target_list
                            [BY attribute]
                            [WITH Pointer]

uc-letter               :=  A | B | C | ... | Z

string                  :=  uc_letter
                            | string uc_letter

lc-letter               :=  a | b | c | ... | z

char_string             :=  uc_letter
                            | char_string lc_letter

digit                   :=  0 | 1 | 2 | 3 | 4 | 5 | 6 |
                            7 | 8 | 9

number                  :=  digit
                            | digit number
```

```
float                   :=  number.number

add_op                  :=  + | -

mult_op                 :=  * | /

expr1                   :=  arith_term1
                            | expr1 add_op arith_term1

arith_term1             :=  arith_factor1
                            | arith_term1 mult_op
                              arith_factor1

arith_factor1           :=  attribute_being_modified
                            | number

expr2                   :=  arith_term2
                            | expr2 add_op arith_term2

arith_term2             :=  arith_factor2
                            | arith_term2 mult_op
                              arith_factor2

arith_factor2           :=  base_attribute
                            | number
```

LIST OF REFERENCES

1. Demurjian, S. A., and others, An Attribute-based System as a Database Kernel of Database Systems, unpublished.

2. Hsiao, D. K., "A Generalized Record Organization," IEEE Transactions on computers, Vol. C-20, No. 12, December 1971.

3. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," Communications of the ACM, September 1971.

4. Hsiao, D. K., and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)," Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, columbus, Ohio, July 1981.

5. Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of National ACM Conference, 1978.

6. Banerjee, J., Buam, R. I. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 4, No. 1, pp. 347-384, December 1978.

7. Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980.

8. Macy, G., Design and Analysis of an SQL Interface for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.

9. Date, C. J., An Introduction to Database Systems, 3d ed., Addison-Wesley, 1981.

10. Hsiao, D. k., and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

11. Astrahan, M. M., and others, "System R: a Relational Approach to Data Management," ACM Transactions on Database Systems, Vol. 1, No. 2, pp. 97-137.

12. Chamberlin, D. D., and Boyce, R. F., "SEQUEL: A Structured English Query Language", Proceedings of ACM SIGFIDET Workshop, Ann Arbor, Michigan, May 1974.

13. Ullman, J. D., Principles of Database Systems, 2d ed., Computer Science Press, 1983.

14. Chamberlin, D. D., and others, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM J. R&D 20, No. 6, November 1976.

15. Muldur, S., The Design and Analysis of Join and Ordering Operations for a Multi-Backend Database System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

16. Weishar, D. J., Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System, Master's thesis, Naval Postgraduate School, Monterey, California, June 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information center Cameron Station Alexandria, Virginia 22314	2
2.	Library, code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
5.	Dr. D. K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6.	Dr. P. R. Strawser, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
7.	Commanding Officer ATTN: LT Griffin N. Macy Naval Security Group Activity Northwest Chesapeake, Virginia 23322	1
8.	Office of the President ATTN: CDR Rich Rollins Naval War College Newport, Rhode Island 02841	2
9.	Robert A. Rollins 8936 Pardee Road Saint Louis, Missouri 63123	2

END

FILMED

4-85

DTIC